Radio Shack
TRS-80
Software Library
™

TRS-80®
MODEL II

An Overview of the TRS-80 Model II Compiler BASIC Manual

The four sections in this manual contain the information you need to use Radio Shack's COMPILER BASIC. We suggest that you begin by running through the steps in the first chapter of Section 1, "Operating Compiler BASIC."

The four sections are:

1/Operating Compiler BASIC

Takes you through the steps of operating Compiler BASIC from starting up the system to typing, debugging, compiling, running, and saving programs. Includes alphabetical entries on each BASIC command.

2/Programming in RSBASIC

Shows you how to write programs using the RSBASIC programming language. Includes alphabetical entries on each BASIC keyword.

3/BEDIT

Explains how to use BEDIT to edit your BASIC source programs.

4/Programmer's Information Section

Gives background information on the Compiler BASIC development system, memory usage, data storage, and assembly language subprograms. Also, gives information on how to use the stand-alone Runtime System.

This manual complements the information in your Model II Operations and TRSDOS manuals. If you need more information on your Model II computer system, we refer you to these manuals.

How Compiler BASIC Works

The BASIC programming language must translate all your BASIC instructions to an object code the computer understands. The means it uses to translate your instructions depends on the form of BASIC you have.

The BASIC which comes with the TRS-80 Model II is an Interpreter. It interprets each instruction to object code everytime it runs the program.

Compiler BASIC, on the other hand, translates the program in two stages. First, it compiles the entire program to an intermediate object code. Then, when running the program, it translates this intermediate code to an object code.

Compiling your program to this intermediate code will give you several advantages:

- The program will take up much less space in memory and on diskette.
- No one using your program will be able to read your "source" BASIC instructions.

Notice To Programmers

By your purchase of the software product described in this book, you have obtained a license to duplicate TRSDOS and Model II BASIC only as necessary for your personal use.

If you intend to sell BASIC applications programs you have written for the TRS-80 Model II, you must follow the procedure below to avoid violation of this license and of the copyright laws.

The complete Radio Shack BASIC Development System (26-4705) includes the TRSDOS™ operating system, the RSBASIC Compiler, the RUNBASIC runtime and numerous auxiliary files.

RSBASIC produces an intermediate code which can only be executed by the runtime system RUNBASIC. Therefore, your compiled program will require that the user have TRSDOS and RUNBASIC from Radio Shack.

Since you may not duplicate TRSDOS or RUNBASIC for resale, you have two options for selling a copy of your own program:

A. Purchase a RUNBASIC/TRSDOS runtime system diskette (Catalog Number 26-4706) from Radio Shack. Copy your compiled program onto this diskette, and sell this diskette to your customer. The copyright notices affixed to that diskette must not be removed or hidden from view. For each copy of your program you sell in this manner, you must purchase the RUNBASIC diskette and copy your program onto it.

B. Sell your compiled program without TRSDOS and without the BASIC runtime. Instruct your customer to purchase a RUNBASIC/TRSDOS runtime from Radio Shack.

The Model II BASIC Interpreter programs are not meant to be run under Compiler BASIC. Radio Shack does not recommend converting BASIC Interpreter programs.

Radio Shack®

Section 1
Operating Compiler BASIC

General Information,
Compiler Use, Start-Up, Commands.

You may use Compiler BASIC in two ways:

1. As a Development System - to write, compile, run, debug, and store programs, or

2. As a Stand-Alone Runtime System - to only run your programs. After developing a program, you might give it to other people to operate by simply using the Runtime System.

This section explains how to use Compiler BASIC as a Development System. For information on the stand-alone runtime system, see the Programmers Information Section. Also see the appendix for information on how to create a runtime system diskette.

We suggest you begin by going through the steps in Chapter 1.

TABLE OF CONTENTS
SECTION 1. OPERATING COMPILER BASIC

```
*****************************************
*                                       *
*              Chapter 1                *
*                                       *
*        USING COMPILER BASIC           *
*                                       *
*****************************************
```

INTRODUCTION

This chapter quickly runs through the mechanics of loading and operating the Model II BASIC Compiler. We only mention certain BASIC commands to illustrate how to operate the Compiler. The details on each command are in the Commands Chapter. Details on the Compiler itself are in the Programmers Information Chapter.

OUTLINE OF CHAPTER 1
USING COMPILER BASIC

Inserting a diskette. Label may
extend vertically across the diskette.

Setting the date and time.

STARTING UP MODEL II COMPILER BASIC

To use this Radio Shack Compiler BASIC package, you will need a TRS-80 Model II with 64K of RAM.

Before loading Compiler BASIC, you need to initialize the Model II disk operating system by setting the date and time. The operating system, called TRSDOS, is on you RSBASIC diskette and is loaded automatically when you insert the diskette.

The Model II operations Manual explains how to connect and power-up the Model II, and how to properly insert a diskette.

SETTING THE DATE AND TIME

As soon as TRSDOS is loaded, it prompts you for the date. Type in the date using the MM/DD/YYYY form and press <ENTER>. For example:

```
     04/01/1980 <ENTER>
```

sets the data for April 1, 1980.

Next, the system prompts you for the time. To skip this question, simply press <ENTER>. TRSDOS starts the clock at 00:00:00.

If you want to set the time, type it in using the 24-hour HH.MM.SS form. You may omit the seconds if you wish. For example:

```
     14.30 <ENTER>
```

starts the clock at 2:30 PM.

The system returns with this message:

```
     TRSDOS READY
     ...........................................
```

At this point you may execute any TRSDOS command or load RSBASIC.

Since all TRSDOS and BASIC commands are capitalized, you'll probably find it convenient to operate the keyboard in the caps mode. Press <CAPS> so the red light comes on. That way, all the alphabet-keys are interpreted as capital letters. When you want to change to the lower case mode, press the <CAPS> key again.

LOADING RSBASIC

The simplest way to load RSBASIC is to type:

```
     RSBASIC <ENTER>
```

After taking a few seconds to load, BASIC displays a start-up heading like this:

```
 TRS-80 MODEL II COMPILER BASIC (RM/BASIC ver 1.0)
(C) 1980 BY TANDY CORP. LICENSED FROM RYAN-McFARLAND CORP.
*..............................................
...............................................
...............................................
................
```

You may now begin programming in BASIC.

Options for Loading RSBASIC

The complete syntax for loading RSBASIC is:

```
RSBASIC filespec T=nnnn, s=xxxx
  'filespec' is a TRSDOS file specification
  'nnnn' is a hexadecimal address representing
      the top memory address accessible by BASIC
  'xxxx' is a hexadecimal address representing the
      size of the stack area to be used by BASIC.
'filespec' T='nnnn' and s='xxxx' are optional
```

This means you have several options you may use in loading RSBASIC:

1. You may load it with an instruction to immediately load and execute a BASIC program. To do this type RSBASIC and the program's file specification. For example:

```
     TRSDOS READY
     RSBASIC FILE:1
```

loads RSBASIC, then loads and executes the program file named FILE from drive 1.

2. You may load it with an instruction to protect high memory for your own object code programs. To do this type RSBASIC followed by T=nnnn (where nnnn is a hexadecimal number representing the top memory address which BASIC may use). For example:

```
    TRSDOS READY
RSBASIC T=E000
```

loads RSBASIC. E000 (decimal 57344) is the highest address BASIC will use.

```
TRSDOS READY
RSBASIC PROG/CMP T=E000
```

Loads RSBASIC and the program PROG/CMP, and immediately executes PROG/CMP. BASIC will not be able to use any memory addresses over E000.

3. You may load it with an instruction to set the stack size to greater than the default stack size of 00C0 (decimal 192) to allow increased usage of BASIC features like GOSUB and CALL, which use more than average amounts of stack space.

```
TRSDOS READY
RSBASIC S=0180
```

loads RSBASIC with a stack size of 0180 (decimal 386).

```
TRSDOS READY
RSBASIC T=E000, S=0180
```

loads RSBASIC with a stack size of 0180 and prevents BASIC from utilizing any memory address over E000.

PROGRAMMING WITH RSBASIC

TYPING THE PROGRAM INTO MEMORY

To type a BASIC program line into memory, type a line number followed by a space followed by a BASIC statement. Unless the line contains 255 characters, you must press <ENTER> to signify the end of the line. This is an example of how to type a program line:

```
   10 PRINT "THIS IS A SAMPLE BASIC PROGRAM LINE" <ENTER>
```

BASIC has six commands to help you in typing and editing a program:

1. AUTO - automatically numbers each program line
2. CHANGE - replaces one group of characters on program lines with another.
3. DELETE - deletes one or more program lines
4. DUPLICATE - duplicates one or more of your program lines in a different part of your program.
5. RENUMBER - renumbers your program.
6. LIST - lists your program.

To use a BASIC command, type the command and then press <ENTER>. For example:

```
     LIST <ENTER>
```

Lists all the program lines you have typed.

Some commands require that you include parameters as part of the command. For example:

```
     CHANGE 10/LINE/
```

changes line 10 by deleting the word LINE. The parameters are 10 and LINE.

The Model II keyboard has certain special keys which are helpful in typing program lines and commands:

<-	Backspaces the cursor without erasing any characters. Use this to position the cursor for correcting a portion of a line.
->	Forward-spaces the cursor without erasing any characters. Use this to position the cursor for correcting a portion of a line.
<BACKSPACE>	Backspaces the cursor, erasing the last character you typed. Use this to correct entry errors.
<ENTER>	Signifies end of line.
<SPACEBAR>	Enters a space (blank) character and moves the cursor one character forward.
<ESC>	Erases the current line. Use this when you want to correct the entire line.
<REPEAT>	For convenience, when you want to repeat a single key, hold down <REPEAT> while pressing the desired key. For example, to backspace halfway back to the beginning of the line, hold down <REPEAT> and <BACKSPACE>.

You may want to use BEDIT to edit your program. The section on BEDIT explains how to do this.

EXECUTING THE PROGRAM

The BASIC Compiler only executes programs which have been compiled into object code. If you are executing a particular BASIC program for the first time, there will be a slight delay before that program is executed in order for BASIC to compile the program.

The BASIC command for executing a program is RUN. To execute this program:

```
      10   PRINT "THIS IS A SAMPLE BASIC PROGRAM"
      20   GOTO 10
```

Type the RUN command:

```
      RUN <ENTER>
```

BASIC compiles and then executes the program. While the program is executing, the Computer is under control of the program. These are the two special keys you may use to interrupt execution of the program:

<HOLD>	Pauses execution of the program. Press again to continue.
<BREAK>	Terminates execution of the program. During line input, the program will wait to terminate execution until you press the <ENTER> key.

Note: RUN does not initialize variable memory during the compiling process. If you are Running the same program a number of times, the program will start each time with the same values it had in variable memory the last time it was Run.

Debugging the Program

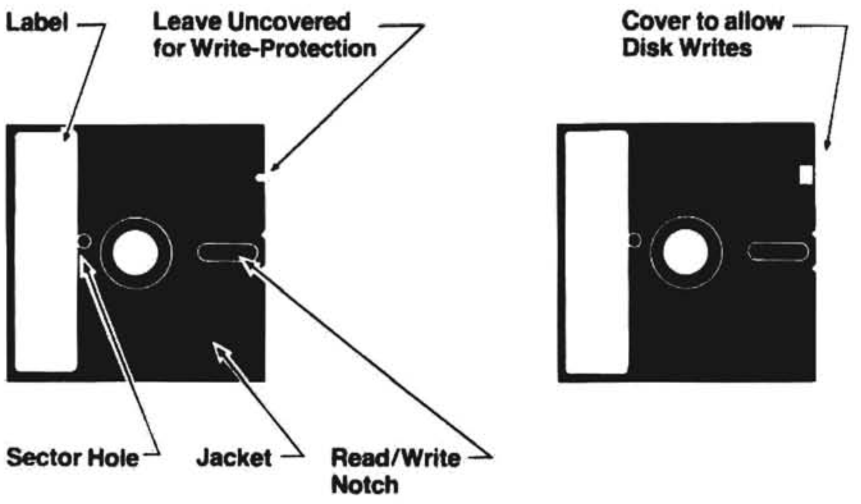
RSBASIC has four commands to help in debugging a program:

1. TRACE - sets up a tracer which displays each line number as it is being executed.
2. BREAK - sets breakpoints in the program which break program execution.
3. STEP - executes a certain number of lines in the program.
4. GO - continues program execution at the next executable statement.

These commands are detailed in the Commands section.

USING THE DISKETTES

You may use diskettes to store any programs or data files you have created. To store data on a diskette, you must cover the write-protect notch on the diskette. Use the gummed tape provided with the diskette.


Label
Leave Uncovered for Write-Protection
Cover to allow Disk Writes
Sector Hole
Jacket
Read/Write Notch

Before using a diskette for storage, make sure the diskette which you want to use is properly inserted. Never insert or remove the diskette while reading or writing to it. This might destroy the contents of the diskette.

ASSIGNING FILE SPECIFICATIONS

Anything you store on diskette must be stored as a disk file with a TRSDOS file specification. Afterwards, you may load the program by specifying the file name you gave to the file when you stored it.

The complete syntax for a file specification is:

```
  filename/ext.password:d
    'filename' is any name up to eight characters
       beginning with a letter.
    '/ext' is an optional extension to the filename
       consisting of up to three characters.
    '.password' is an optional password with up to
       eight characters.
    ':d' is an optional drive specification (0,1,2, or 3).
    You may use this if you have a multi-drive system
       to specify which disk drive you want to use in
       saving and loading the program.
```

Only 'filename' is essential. Both '/ext' (extension) and '.password' are optional extensions which you may add to the filename. ':d' is also optional. If you have a multi-drive system, it specifies which drive you are using for storage.

Examples of file specifications:

```
    BOOK/BAS.ABCDE:2
```

The filename is BOOK, the extension to the filename is BAS, the password is ABCDE. The diskette in drive number 2 will be used in saving or loading the program.

```
    PROGRAM
```

The filename is PROGRAM. There is no extension, password, or drive specification. Since there is no drive specification, BASIC will use the first available drive beginning with drive 0 (the built-in drive).

```
    ACCOUNT1/CMP:1
```

The filename is ACCOUNT1. The extension is CMP. The diskette in drive number 1 will be used in saving or loading the program.

```
    PAYROLL.SECRET
```

The filename is PAYROLL. The password is SECRET. There is no extension to the filename and no drive specification.

Note: See Section 1 in the Model II Disk Operating System Manual for more information on TRSDOS file specifications.

STORING A PROGRAM ON DISKETTE

RSBASIC has two commands for storing a program on diskette: SAVE and COMPILE. The SAVE commands stores the program in its existing BASIC format. COMPILE compiles the program to object code and saves it as an object code program.

Saving a Program:

To SAVE a program which is currently in memory, simply type the SAVE command followed by the file specification you are assigning to the program. For example, to save this program (once it has been typed into memory):

```
     10  PRINT "THIS IS AN EXAMPLE OF A BASIC PROGRAM"
     20  GOTO 10
```

You may type:

```
     SAVE EXAMPLE/BAS <ENTER>
```

This gives the program the file name EXAMPLE, with the extension BAS, and saves it on the diskette in drive 0 -- the built in drive. (If you have a multi-drive system, RSBASIC will save it on the first diskette available,beginning its search with the diskette in drive 0).

A Note of Caution

If you save a file with the same file specification as an existing file, the contents of the existing file will be destroyed. For instance, if you save another program under the name EXAMPLE/BAS, the program file you just created above will be destroyed in order to make room for the new file.

For this reason, you might want to check the diskette's directory to see what files are already on the diskette before executing the SAVE command. To do this, type:

```
     SYSTEM "DIR" <ENTER>
```

This executes the TRSDOS command DIR, which displays the contents of the diskette in drive 0.

```
     SYSTEM "DIR:2" <ENTER>
```

Displays the contents in drive 2.

For more information, see SYSTEM in the Commands Chapter of this manual and DIR in the Model II Disk Operating System Manual.

Compiling a Program

Now that the program above is saved as a BASIC program, you may compile it to an object code disk file. Type:

```
     COMPILE EXAMPLE/BAS, EXAMPLE/CMP <ENTER>
```

This compiles the program disk file named EXAMPLE/BAS and stores it on diskette as an object code file with the name EXAMPLE/CMP. The original source program is left unchanged. You should be sure to save it in case you ever need to modify the program (see below).

There are several reasons for compiling a long program:

1. The compiled program takes up less room, both on diskette and in memory.
2. Once you have a program in final form, so that further editing and debugging is not required, you don't need all the overhead of the RSBASIC Development System. Instead, you may copy the compiled program onto a diskette containing only the RUNBASIC program. This leaves maximum disk space available for your data files.

You cannot edit, list or otherwise modify a compiled program. If you ever need to modify it, you simply edit the original source program and re-compile it.

CLEARING MEMORY

Once programs are saved on diskette, you will probably want to clear the Computer's memory. BASIC has two commands for this:

1. NEW - erases all BASIC programs from memory but keeps compiled object code programs in memory.
2. CLEAR - erases all BASIC and compiled programs from memory, undefining all variables.

For example, to erase all programs from memory, type:

```
     CLEAR <ENTER>
```

LOADING PROGRAMS FROM DISK

BASIC has different commands for loading BASIC and Compiled programs from diskette.

Loading a BASIC Program

The OLD command loads a BASIC program from diskette. For example:

```
     OLD EXAMPLE/BAS
```

Loads the program from diskette named EXAMPLE/BAS, which was stored above with the SAVE command. Once the program is loaded, you may execute it with the RUN command.

Since memory is cleared everytime you OLD a program, BASIC offers two commands to use in loading more than one BASIC program: APPEND and MERGE.

Loading a Compiled Program

The LOAD command loads Compiled programs from diskette. For example:

```
     LOAD EXAMPLE/CMP <ENTER>
```

Loads the program from diskette named EXAMPLE/CMP, which was stored above with the COMPILE command. Once loaded, the program may be executed with RUN.

Unlike OLD, LOAD does not clear memory when it loads a program. Therefore, you may load a series of Compiled programs into memory.

STORING DATA FILES ON DISKETTES

To store data files on diskette, see the chapter on Data Files.

Using This Package with the Model II Hard Disk (26-4150)

This software package can be used with the TRS-80 Model II Hard Disk System.

However, before FCOPYing this package over to TRSDOS-HD (see the Hard Disk Owner's Manual), make the following program modifications:

```
PATCH RSBASIC/LIO A=377D F=FD5609FD5E08 C=014700090000
PATCH RSBASIC/LIO A=3786 F=ED53 C=0022
PATCH RSBASIC/OLF R=156 B=126 F=FD5609FD5E08 C=014700090000
PATCH RSBASIC/OLF R=156 B=135 F=ED53 C=0022
```

Then transfer the program has described in the Hard Disk Owner's Manual (see FCOPY). Once the program has been transfered, you may use it as described in this manual.

```
*******************************************
*                                         *
*             Chapter 2                   *
*                                         *
*             COMMANDS                    *
*                                         *
*******************************************
```

INTRODUCTION

Compiler BASIC is made up of commands. These commands instruct it to do something immediately.

In this chapter, there are alphabetical entries for each command. The next two pages explain the format for each command. On the following page is a brief introduction to commands.

OUTLINE FOR CHAPTER 2 COMMANDS

FORMAT FOR COMMAND ENTRIES

(1) 1. The first line is the command itself. The second line briefly describes what it does.

(2) 2. The information in the gray box is the syntax for the command. The first line shows the format to use in typing the command. This format line always contains:

a. the command itself

and may also contain:

b. parameters

c. options

If the syntax contains parameters and options, the next lines define them. A parameter enclosed in single quotes indicates that you must specify its value. In the syntax illustrated here, you must specify 'startline' and 'endline', if you choose to use these parameters.

(3) 3. This paragraph explains how to use the command.

(4) 4. These examples illustrate how the command might be used.

-- COMMAND --

LIST
Display Program Lines 1

```
   LIST startline-endline string A {PRT}
     'startline' is a line number specifying the lower
         limit for the listing.
     'endline' is a line number specifying the upper limit
         for the listing.  If omitted, only 'startline'
         will be listed.
     'string' is a string constant or a string variable.
         If A is omitted, only the first statement which         2
         contains 'string' will be listed.  'string' A may
         be omitted .
     PRT causes the listing to appear on the line printer
         rather than the video display.
      Note:  if both 'startline' and 'endline' are omitted,
      the entire program will be listed.
```

The LIST command gets the Computer to display a program line or a group of program lines that are currently in memory. If you do not specify any line numbers with the LIST command, it will list all the lines. You can use the PRT option to cause the listing to be printed on the line printer. 3

You may specify a certain string you would like listed by putting it between any two non-numeric delimiting characters except " - ".

Examples

```
   LIST
```

Displays the entire program. To stop the automatic scrolling, press <HOLD>. This will freeze the display. Press <HOLD> again to continue the listing. 4

```
   LIST 50
```

INTRODUCTION TO COMMANDS

A command instructs the Computer to immediately do something. For example:

```
     *LIST <ENTER>
```

instructs the computer to immediately display all program lines currently in memory. A command may not be part of the program.

All BASIC commands may be abbreviated by the first two letters in the command. For example, LIST may be abbreviated by:

```
     *LI <ENTER>
```

You may specify certain parameters for some of these commands. For example:

```
     *LIST 50-80
```

instructs the computer to immediately list lines 50 through 80. The parameter is 50-80.

When typing a command with a parameter, there must be a space or a comma after the command. This, for example would produce an error:

```
     LIST50-80
```

A few of the commands also include options:

```
     *LIST 50-80 {PRT}
```

lists lines 50-80 on the line printer. The option is {PRT}. Options may always be omitted from the command if you don't want to use them.

-- COMMAND --

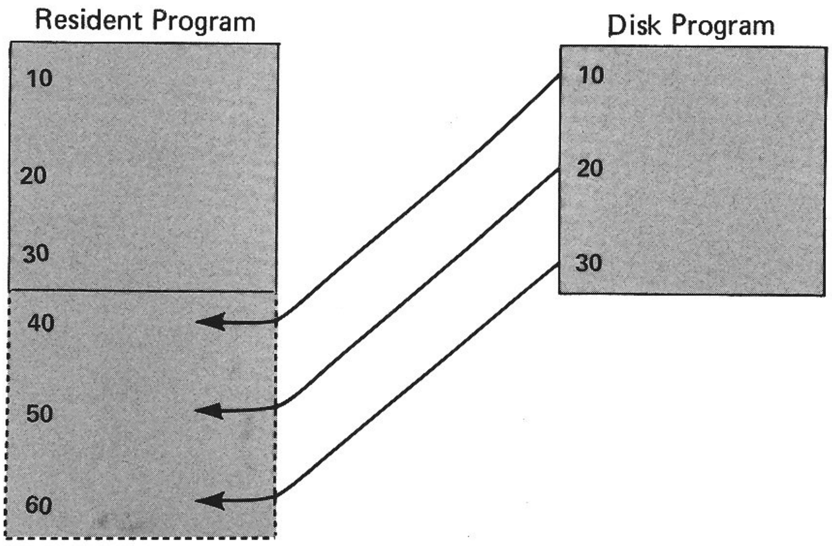
APPEND
Append Two Programs

```
APPEND file
   'file' is a TRSDOS file specification for
       a BASIC source program.
```

APPEND joins a program from disk to the resident program. The appended disk program is renumbered to follow the resident program. Its first renumbered line is computed by adding ten to the last line number of the resident program. Ten is added to each successive line.

While the program is being appended, you may stop this process by pressing <BREAK>. The original resident program will be restored.

Only source programs can be appended. You can not use APPEND to append an object program from disk which was created with the COMPILE command.



Examples

```
   APPEND PART2/BAS:1
```

This loads the program PART2/BAS from drive 1. It is renumbered to follow the resident program.

```
   APPEND PROG2
```

PROG2 is appended to the resident program. Since no drive is specified, BASIC will begin searching for it in drive 0.

```
   AP GRAPH/SUB
```

The subprogram GRAPH/SUB is appended to the main program in resident memory.

--COMMAND --

AUTO
Number Lines Automatically

```
AUTO startline, increment
   'startline' is a line number specifying the first
      line number to be used.
   'increment' is a number specifying the increment
      to be used between lines.  If increment
      is omitted, 10 is used.
   If both 'startline' and 'increment' are omitted,
      startline will be the last line plus 10 and
      increment will be 10
```

The AUTO command helps you type program lines faster by automatically numbering each line. To use it, type AUTO, then type the number you want as your first automatic line number (startline), and then, finally, type the number of lines you want between each program line (increment).

After you type this command and press <ENTER>, BASIC will supply you with the first line number. All you have to do is type in your program statement and press <ENTER>. BASIC will then supply the next line number.

To turn off AUTO, press <ENTER> after AUTO displays a line number. If AUTO supplies you with a line number that has an asterisk beside it, this means you have already used this program line. Press <ENTER> if you do not want to change the line.

Examples

```
   AUTO
```

If you have not typed any program lines yet, this will start automatic line numbering with line 10. If you have typed any program lines, automatic line numbering will start at 10 plus the last program line. This command increments each line number by 10.

```
   AUTO 100
```

starts numbering with 100, using increments of 10 between line numbers.

```
   AUTO 1000, 100
```

starts numbering with 1000, using inrements of 100 between line numbers.

```
   AU 5
```

starts numbering with 5 using increments of 10 between line numbers.

-- COMMAND --

BREAK
Set or Remove Program Breakpoints

```
BREAK line number, ...
   if 'line number' is omitted, all breakpoints will be
      cleared.
```

BREAK sets a certain line or series of lines as a breakpoint in the program. When BASIC encounters this line it will stop executing the program and return to the command mode. This will happen before the breakpoint line is executed. Use the GO command to continue program execution.

You can set more than one breakpoint. To clear all the breakpoints, use BREAK without any line numbers.

Examples

```
   BREAK 120
```

When the program is run, BASIC will stop execution and enter the command mode immediately before line 120.

```
   BREAK 200, 300, 400
```

This sets lines 200, 300, and 400 as breakpoints. BASIC will stop program execution when it encounters any of these lines. The GO command continues program execution to the next breakpoint or to the end of the program.

```
   BR
```

This clears all the breakpoints. The program will execute normally.

-- COMMAND --

CHANGE
Change Program Lines

```
CHANGE startline-endline del oldstring del
newstring del A
  'startline' and 'endline' are line numbers specifying
      the lower and upper limits of program lines
      that will be changed.  If 'endline' is omitted,
      only 'startline' will be changed.  If both
      'startline' and 'endline' are omitted,
      the entire program will be changed.
  'oldstring' and 'newstring' are string constants
  'del' is any non-numeric character other than "-".
  if A is omitted, only the first occurrence of
      'oldstring' in a program line will be changed.
```

CHANGE edits program lines by replacing the oldstring with the newstring. CHANGE, of course, can only be used on source programs which are in their original BASIC form.

Examples

```
CHANGE 100-200/PRINT/LPRINT
```

The first occurrence of "PRINT" in all lines is changed to "LPRINT". Notice that since the A option is not used, only the first occurrence is changed. In this example, slashes are used as delimiters, although any other character besides the hyphen could have been used.

```
CHANGE,TAB(10),TAB(5),A
```

Every occurrence of "TAB(10)" is replaced by "TAB(5)" in all of the lines. Commas are used here as delimiters.

```
CHANGE 500-1000/REM/
```

The first occurrence of "REM" in all lines from 500 to 1000 is changed to the null string; i.e., deleted.

```
   CH 100/JOHN ANDERSON/JAMES KNIGHT
```

Changes the first occurrence of "JOHN ANDERSON" in line 100 to "JAMES KNIGHT".

-- COMMAND --

CLEAR
Clear All Programs from Memory

```
CLEAR
```

When CLEAR is used, all programs are deleted from memory, all variables are undefined, and the system is returned to its initial state. Unlike NEW, CLEAR will also delete compiled object programs from memory.

Example

```
   CLEAR
```

All programs presently in memory are cleared. All variables are undefined.

-- COMMAND --

COMPILE
Compile BASIC Program

```
COMPILE "source file", "object file" {LIST, PRT,
PRT=listing file, MAP, XREF}
  'source file' and 'object file' are TRSDOS file
    specifications.
    'source file' is a BASIC source program file
    'object file' is the object program file that
         COMPILE will create
   All the options below may be omitted:
     LIST generates a source listing containing the
       module relative location of every statement.
     PRT causes all listings to be printed on the
       line printer.
     PRT = listing file routes the printer-formatted
       listing to the specified file.
     MAP generates a memory map showing the location of
       each variable in the program
     XREF prints a cross reference of every reference
       to every variable in the program.
```

COMPILE translates and saves a BASIC program on disk as an intermediate object code program. Once a program is compiled, it is no longer a BASIC program. It cannot be changed.

For this reason, it is adviseable to keep a disk copy of your BASIC source program file until you are sure that you will not want to revise it any more.

There are several advantages to having a compiled disk copy of your BASIC program:

1. The compiled program takes up less room, both on diskette and in memory.

2. If you will be using the stand-alond Runtime System (described in the Programmers Information Section) to run your program, the program must be compiled.

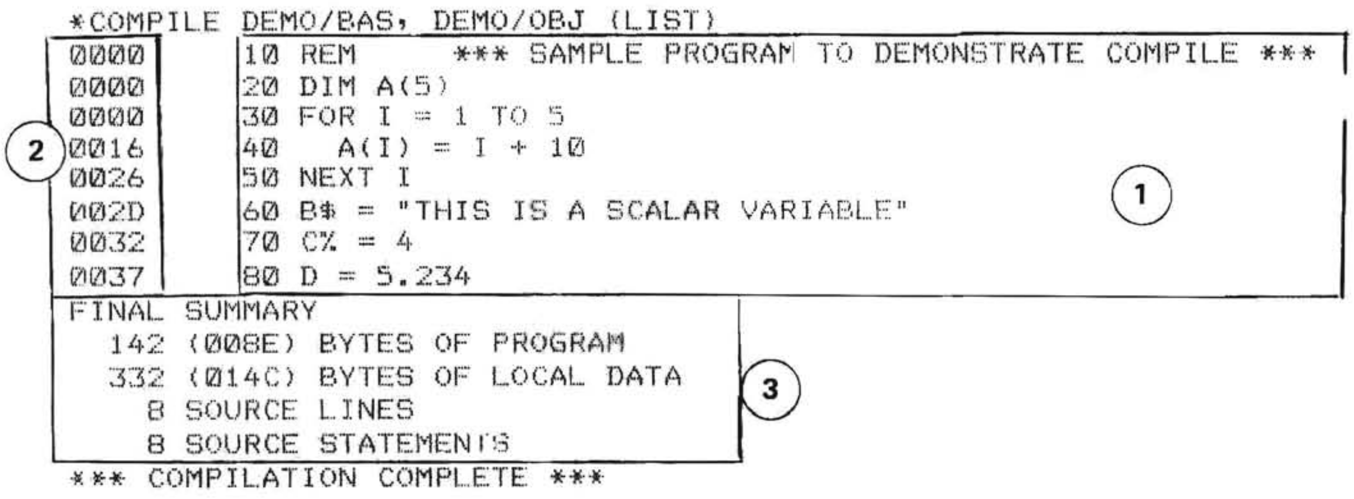
To compile a BASIC program, follow this procedure:

1. use the SAVE command to save your BASIC source program file on disk. Then you may ...
2. use the COMPILE command to create an object code program file on disk from the BASIC source program file.

If the file name you assign to the compiled program already exists, the existing file's contents will be wiped out. It will be replaced by your program.

COMPILE can be used with four options:

A. LIST generates a listing of the program containing the relative memory location of every statement. In the listing below:



```
*COMPILE DEMO/BAS, DEMO/OBJ (LIST)
 0000      10 REM      *** SAMPLE PROGRAM TO DEMONSTRATE COMPILE ***
 0000      20 DIM A(5)
 0000      30 FOR I = 1 TO 5
 0016      40    A(I) = I + 10
 0026      50 NEXT I
 002D      60 B$ = "THIS IS A SCALAR VARIABLE"
 0032      70 C% = 4
 0037      80 D = 5.234
 FINAL SUMMARY
   142 (008E) BYTES OF PROGRAM
   332 (014C) BYTES OF LOCAL DATA
     8 SOURCE LINES
     8 SOURCE STATEMENTS
*** COMPILATION COMPLETE ***
```

1. the source program is displayed
2. the relative memory location of each statement is displayed in hexadecimal notation. For instance, if the program originates at memory location hex 4000, the code for the statement in line 40 would begin at location hex 401A.
3. the final summary displays that the entire program uses 142 bytes of memory. The variables in the program uses 332 bytes.

B. MAP shows the hexadecimal memory location of the variables in the program. in the example below:

```
*COMPILE DEMO/BAS, DEMO/OBJ (MAP)
SYMBOLIC MEMORY MAP
SCALARS
ØØ78     B        STRING*255       ØØAØ     C         INTEGER
ØØA2     D        REAL             ØØ8E     I         REAL
ARRAYS
ØØ7Ø     A(5)               REAL
```

the program contains four scalars (simple variables) and one array variable. In this example B is a string variable containing 255 bytes. It is stored beginning at location hex 0078. A is an array of real numbers containing five elements beginning at location hex 0070.

C. XREF generates a cross reference listing. Each variable is cross referenced with all the line numbers which referenced it. In the example below:

```
*COMPILE DEMO/BAS, DEMO/OBJ [illegible]
CROSS REFERENCE LISTING

SCALARS
B                   6Ø
C                   7Ø
D                   8Ø
I                   [illegible]          [illegible]       [illegible]
ARRAYS
A
```

.. I is referenced on lines 30, 50, and twice on line

D. PRT causes any of the above listings to be listed on the line printer.

E. PRT= listing file causes the listing to be saved in the specified disk file.

Examples

```
   COMPILE BILLING/BAS:0, BILLING/CMP:1
```

The program BILLING/BAS in drive 0 is compiled and saved as a pseudo code program named BILLING/OBJ on the disk in drive 1.

```
    COMPILE BASIC, OBJECT
```

The program BASIC is compiled and saved as a pseudo code program named OBJECT.

```
    COMPILE PAYROLL/BAS, PAYROLL/CMP {LIST, PRT}
```

The source program PAYROLL/BAS is compiled and saved on disk a the pseudo code program PAYROLL/CMP. A listing showing relative memory locations is printed on the line printer.

```
    CO ENTRY/BAS, ENTRY/CMP {MAP, XREF}
```

BASIC compiles this file and displays a memory map and a cross reference listing.

```
    COMPILE PROG/BAS, PROG/CMP {LIST, PRT=FILE}
```

PROG/BAS is compiled and saved on disk as a pseudo-code program named PROG/CMP. A listing is printed and is also saved on disk in a file named FILE.

-- COMMAND --

DELETE
Erase Program Lines from Memory

```
DELETE startline-endline
   'startline' is an existing program line number
       specifying the lower limit for deletion.
   'endline' is an existing program line number
       specifying the last line in your program
       that you want to delete.  'endline' must
       reference an existing program line.
       If omitted, only 'startline' will be deleted.
```

DELETE removes one or more program lines from memory. Another way to delete one program line is to simply type the line number and press <ENTER>.

Examples

```
   DELETE 70
```

Erases line 70 from memory. If there is no line 70, you will get an error message.

```
   DE 50-110
```

Erases lines 50 through 110, inclusive.

```
   70
```

Erases line 70.

-- COMMAND --

DISPLAY
Display Variable Contents

```
DISPLAY subname; variable list, subname; variable
  name...
   'subname' is the name of a subprogram. If
        omitted, the variable contents of the main
        program will be displayed.
```

This command displays the contents of variables in the resident source program. To display the contents of a subprogram's variables, you must specify the name of the subprogram.

All variables are undefined until the program has been compiled. Therefore, you must compile the program first by executing it before using the DISPLAY command.

Examples

```
DISPLAY A
```

Displays the contents of variable A in main memory.

```
DISPLAY A,B$
```

Displays the contents of variables A and B$ in main memory.

```
DI SUBPROG; X
```

Displays the contents of variable X in the subprogram named SUBPROG.

```
DI SUBPROG; X, Y
```

Displays the contents of variable X in SUBPROG and variable Y in the main program or subprogram being executed.

-- COMMAND --

DUPLICATE
Duplicate Program Statements

```
DUPLICATE startline-endline, new startline
   'startline' and 'endline' are the lower and upper
      boundaries of the lines you want to duplicate.
      If 'endline' is omitted, only 'startline' will
      be duplicated.
   'new startline' is the program line which you want
      the duplicated lines to follow. 'New startline'
      must be a current program line.
```

DUPLICATE copies existing program statements to another area of the program. The duplicated program statements begin at 1 + the current program line number you specify. Each successive line number is incremented by one. DUPLICATE does not change any of the existing program statements.

If BASIC must wipe out an existing program statement to duplicate a statement in the area of the program that you specify, it will give you an error message.

As with all editing commands, this command may not be used on a compiled object code program.

Examples

```
   DUPLICATE 100-150, 300
```

The statements in line numbers 100-150 are copied. The duplicated statements appear on line numbers 301, 302, with each additional line number incrementing by 1 until all the statements are copied.

```
   DU 100, 50
```

The statement on line 100 is copied and appears on line 51.

-- COMMAND --

GO
Start or Continue Program Execution

```
   GO
```

GO continues execution of the program after a breakpoint has been encountered. (See BREAK and STEP for information on how to set the break program execution). The GO command can also be used at the beginning of a program to start program execution.

Example

```
   GO
```

Starts or continues executing the program.

-- COMMAND --

KILL
Delete File from Disk

```
KILL file
  'file' is a TRSDOS file specification.
```

KILL deletes the file you specify from the diskette directory. You may Kill a file you will not use again to make room for storing another file.

If you do not specify a disk drive in the file specification, BASIC will search for the first drive that contains the file, and delete it.

Make sure that you do not Kill an open file. If you have used the OPEN statement to open a file, close it before Killing the file.

Examples

```
KILL FILE/BAS
```

deletes FILE/BAS from the diskette in the first drive that contains it.

```
KILL DATA:2
```

deletes DATA from the diskette in drive 2 only.

-- COMMAND --

LIST
Display Program Lines

```
LIST startline-endline string A {PRT}
  'startline' is a line number specifying the lower
      limit for the listing.
  'endline' is a line number specifying the upper limit
      for the listing.  If omitted, only 'startline'
      will be listed.
  'string' is a string constant or a string variable.
      If A is omitted, only the first statement which
      contains 'string' will be listed.  'string' A may
      be omitted .
  PRT causes the listing to appear on the line printer
      rather than the video display.
   Note:  if both 'startline' and 'endline' are omitted,
   the entire program will be listed.
```

The LIST command gets the Computer to display a program line or a group of program lines that are currently in memory. If you do not specify any line numbers with the LIST command, it will list all the lines. You can use the PRT option to cause the listing to be printed on the line printer.

You may specify a certain string you would like listed by putting it between any two non-numeric delimiting characters except " - ".

Examples

```
   LIST
```

Displays the entire program. To stop the automatic scrolling, press <HOLD>. This will freeze the display. Press <HOLD> again to continue the listing.

```
   LIST 50
```

Displays line 50

```
   LIST 50-85
```

Displays lines 50 through 85, inclusively.

```
   LIST 50 {PRT}
```

Prints line 50 on the line printer.

```
   LIST 50-85 {PRT}
```

Prints lines 50 through 85, inclusively, on the line printer.

```
   LIST "PRINT" A
```

Lists all statements which contain the word PRINT

```
   LI/INSERT/
```

Lists the first statement which contains the word "INSERT".

```
   LI  50-80/INSERT/A {PRT}
```

Lists all statements between line 50 and line 80, inclusively, which contain the word INSERT, on the line printer.

-- COMMAND --

LOAD
Load Compiled BASIC Programs

```
LOAD file
   'file' is a TRSDOS file specification for a
       compiled object code program.
```

The LOAD command is used to load compiled programs, which were stored on disk using the COMPILE command, into memory. It will only load object code programs. Use OLD to load BASIC source programs from disk which were stored with the SAVE command.

LOAD can be used to load main programs or subprograms. Since LOAD does not clear resident programs, more than one program can be loaded before executing them. The loading process links the programs together.

Examples

```
   LOAD PROG1/CMP:2
```

This loads PROG1/CMP from drive 2.

```
   LOAD PROG1/CMP
```

Since no drive specification is included in this command, BASIC will begin searching for this program file, starting with drive 0.

```
   LO SUBPROG/CMP:1
```

BASIC loads this subprogram from drive 1.

-- COMMAND --

MERGE
Merge Disk Program with Resident Program

```
MERGE file
   'file' is a TRSDOS file specification for a BASIC
      source file.
```

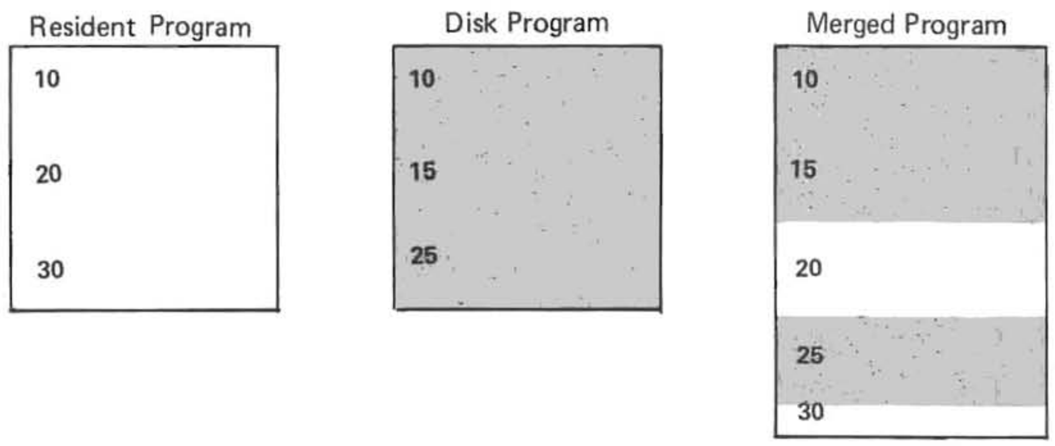
You can use the MERGE command to merge two BASIC source programs into one. MERGE takes a BASIC source program from disk and merges it with the BASIC program you presently have resident in memory.

Both programs must be BASIC source programs. You may not Merge compiled programs.

The program lines from the disk program are merged into the resident program. For an example of how this works, say the disk program contains line numbers 75, 85, and 90. The main program contains lines 70, 80, and 100. When MERGE is used on the two programs, the new program will be numbered 70, 75, 80, 85, 90, 100.

If the line numbers on the disk program coincide with the resident program, the resident lines will be replaced by the disk program. For example, if the disk program is numbered 5, 10, and 20, and the resident program is numbered 10, 20, and 30, the Merged program will be numbered 5, 10, 20, 30. Lines 10 and 20 of the new program will be identical to lines 10 and 20 on the disk program.

MERGE closes all files and deletes all variables.



Examples

    MERGE PROG

This merges the BASIC source program on disk na... ..OG with whatever BASIC program is resident in memory.

    ME PROG/BAS:1

This merges PROG/BAS from the disk drive number 1 with the BASIC program resident in RAM.

-- COMMAND --

NEW
Erase BASIC Program from Memory

```
NEW
```

NEW erases an entire BASIC source program from memory.

NEW does not erase a compiled program which was loaded with the LOAD command.* Use CLEAR to erase all programs from memory.

*NEW will erase a compiled program which was loaded with the RUN command.

Example

```
NEW
```

Sample Use

NEW can be very helpful when you want to erase your main BASIC program, but would like to keep your compiled subprograms in memory to use with your next BASIC program. By executing the command:

```
NEW
```

Your main BASIC program is erased from memory, but all object programs remain. You may now load or type in another BASIC program to use with your compiled subprograms.

-- COMMAND --

OLD
Load BASIC Source Program

```
OLD file
  'file' is a TRSDOS file specification for a
      BASIC source program file
```

The OLD command loads a BASIC source program, saved on disk, into RAM. OLD will only load BASIC source programs. Use LOAD to load a compiled program.

Since OLD clears all resident BASIC programs before loading a program, only one BASIC program may be loaded into memory with this command. To get other BASIC programs into memory, use MERGE or APPEND.

Examples

```
   OLD PROG/BAS:2
```

Loads PROG/BAS into RAM from drive 2.

```
   OL PROG/BAS
```

Loads PROG/BAS into RAM. Since no drive specification is included, BASIC will begin searching for it in drive 0.

-- COMMAND --

RENUMBER
Renumber Program

```
RENUMBER newline, increment
   'newline' specifies the new line number of the first
      line to be renumbered.
   'increment' specifies the increment to be used
      between each successive renumbered line.  If
      'increment' is omitted, 10 is used.
   If both 'newline' and 'increment' are omitted, 10
      is used for newline and 10 for increment.
```

RENUMBER changes all the line numbers in your program. It also changes all line number references appearing after GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, and ON ERROR GOTO.

Examples

```
   RENUMBER
```

Renumbers the entire resident program. The first new line number is 10 and each line is incremented by 10.

```
   RENUMBER 6000, 100
```

Renumbers the program. The first new line number is 6000 and each line is incremented by 100.

```
   RE 10000
```

Renumbers the program. The first new line number is 10000 and each line is incremented by 10.

-- COMMAND --

RUN
Execute Program

```
RUN file
  'file' is a TRSDOS file specification.  It may
      be a BASIC source program file or an object
      code program file.  If omitted, the resident
      program will be run.
```

RUN is the command that executes your program. RUN compiles, if necessary, and executes the program that is in resident memory. If the program is in the form of a BASIC source program, there will be a short delay while RUN is compiling the program before running it.

If you include a file specification, BASIC will load or old the program from disk and execute it. You may have BASIC Run either a BASIC source program or a compiled program. If you use RUN to run a compiled program, be sure to first clear any BASIC programs you have in resident memory.

Examples

```
RUN
```

Executes the program in resident memory.

```
RUN PROGRAM/CMP:2
```

Loads the compiled program PROGRAM/CMP from drive 2 and executes it.

```
RUN PROGRAM/BAS
```

Loads the BASIC source program PROGRAM/BAS and executes it.

```
RU PROGRAM
```

Loads the program PROGRAM and executes it.

-- COMMAND --

SAVE
Save BASIC Source Program on Disk

```
SAVE file
   'file' is a TRSDOS file specification.  If
      omitted, the program will be saved under
      the file specification used in the last
      OLD command.
```

BASIC has two commands for storing programs on a disk file: SAVE and COMPILE. SAVE stores the program in its existing BASIC source program format. COMPILE converts the program and stores it as an intermediate object code program.

SAVE is the best command to use when storing programs that you might list, revise, or add to in the future. To use it type SAVE and the appropriate file specification. (See the section on TRSDOS file specifications).

If you SAVE a program using a file specification that already exists, the existing program file will be wiped out. It will be replaced by the program file you are saving.

You may leave out the file specification with SAVE. The program will then be saved under the same file specification that you used to load the last program with the OLD command.

To label the files that are BASIC source programs versus the Compiled object programs, we suggest you use the extension /BAS for Saved programs and /CMP for Compiled programs.

A Saved program is in ASCII code or text format.

Examples

```
   SAVE FILE1/BAS.JOHNQDOE:3
```

Saves the resident BASIC program. The filename is FILE1, the extension is /BAS, and the password is JOHNQDOE. The file is stored on the disk in drive 3.

```
SAVE FILE1/BAS
```

Saves the resident BASIC program. The filename is FILE1 and the extension is /BAS. Since no drive is specified, BASIC will store the program in the first drive which has room for it.

```
   SA
```

Saves the resident BASIC program. It will be saved under the same file specification used in the last OLD command.

-- COMMAND --

SIZE
Print Used and Unused Memory

```
   SIZE
```

By executing the SIZE command, BASIC will print the amount of space being used by the resident program and the amount of space that is unused. The values are expressed in bytes both as a decimal and a hexadecimal value.

Example

```
   SIZE
```

Prints the number of bytes the resident program is using, and the number of unused bytes remaining in memory.

-- COMMAND --

STEP
Execute Portion of Program

```
STEP number
   'number' is the number of lines to execute
```

STEP executes the number of lines in the program you specify, beginning with the next executable statement.

STEP is normally used in debugging a program. You may execute the entire program portions at a time using STEP.

Example

```
     STEP 5
```
Executes the next five statements in the program.

-- COMMAND --

SYSTEM
Return to TRSDOS

```
SYSTEM del "command"
  'del' is a comma or a space.
  'command' is a TRSDOS command.  If 'command' is
     omitted, the system will return to TRSDOS READY.
```

SYSTEM returns you to TRSDOS, the disk operating system. If you specify a TRSDOS command, your system will execute the command and immediately return to BASIC. Your program and variables will remain intact.

If you do not specify a command, SYSTEM will return you to the TRSDOS READY mode.

Examples

```
   SYSTEM
```

Returns you to TRSDOS READY. Your resident BASIC program will be lost.

```
   SYSTEM "DIR"
```

Executes the TRSDOS command DIR (print directory), and then returns to BASIC. Your resident BASIC program will remain intact.

-- COMMAND --

TRACE ON, TRACE OFF
Turn Tracer On, Off

```
   TRACE ON
   TRACE OFF
   TRACE
```

TRACE is a useful command for debugging and analyzing a program. TRACE ON turns on a tracer. Each time the program advances to a new program line, the line number will be displayed.

TRACE OFF turns the tracer off. TRACE prints whether the tracer is on or off.

Examples

```
   TRACE ON
```

When the program is RUN each program line number will be printed in while that line is executing.

```
   TR OFF
```

Turns off the tracer.

```
   TRACE
```

Prints whether the tracer is on or off.

Radio Shack®

Section 2 Programming with RSBASIC

Information on writing a program with RSBASIC.

Compiler BASIC supplies the language RSBASIC to use in writing programs. RSBASIC is a form of BASIC, and in this manual, we refer to it as BASIC. This section has the reference information you need to use RSBASIC.

We are assuming that you are already familiar with BASIC. If you are a newcomer to BASIC, there are many good BASIC teaching books available. Here are some we recommend:

COMPUTER PROGRAMMING IN BASIC FOR EVERYONE, Thomas Dwyer and Michael Kaufman, Radio Shack Catalog Number 62-2015.

BASIC AND THE PERSONAL COMPUTER, Thomas Dwyer and Margot Critchfield; Addison-Wesley Publishing Company, 1978.

BASIC FROM THE GROUND UP, David E. Simon; Hayden Book Company, 1978.

ILLUSTRATING BASIC, Donald Alcock; Cambridge University Press, 1977.

TABLE OF CONTENTS

SECTION 2. PROGRAMMING WITH RSBASIC

SPECIAL MODEL II PROGRAMMING TIPS

Programming the Video Display

The Model II Video Display has two modes: scroll and graphics. With the exception of graphics characters, BASIC prints all output to the display using the scroll mode. See PRINT for information on programming in the scroll mode. See CRTG for information on programming in the graphics mode. (Both PRINT and CRTG are in the Keywords Chapter).

Programming the <F1> and <F2> Keys

You may program the <F1> and <F2> keys to be interpreted a certain way in your program through using the ASC function. ASC converts a character to its ASCII code. The ASCII code for the <F1> key is 1; the code for <F2> is 2.

It is easier to program the <F1> and <F2> keys using character input functions (INKEY$ and INPUT$) rather than line input statements (INPUT and LINE INPUT).

```
 10  PRINT "PRESS <F1> TO ENTER NEW ACCOUNTS"
 20  PRINT "PRESS F2> TO REVISE ACCOUNTS"
 30  A$ = INPUT$(1)
 40  IF ASC(A$) = 1 THEN 100
 50  IF ASC(A$) = 2 THEN 200
 60  STOP
100  PRINT "ACCOUNTS ENTRY PROGRAM"
110  STOP
200  PRINT "ACCOUNTS REVISION PROGRAM"
```

For more information, see ASC, CHR$, INPUT$, and INKEY$ in the Keywords Chapter.

```
**************************************************
*                                                *
*                   Chapter 3                    *
*                                                *
*                 BASIC Concepts                 *
*                                                *
**************************************************
```

INTRODUCTION

This chapter explains how BASIC handles and manipulates data. This information will prove helpful in writing programs which handle data more efficiently.

OUTLINE OF CHAPTER 3
BASIC CONCEPTS

OVERVIEW -- ELEMENTS OF A PROGRAM

PROGRAM

A program is made up of one or more numbered lines. Each line contains one or more BASIC statements. BASIC allows line numbers from 0 to 65535 inclusive. The maximum number of lines BASIC allows in a program are 2048 lines.

You may include up to 255 characters per line, not including the line number. You may also have two or more statements to a line, separated by colons.

Here is a sample program:

```
 line         BASIC         colon between       BASIC
number       statement     statements          statement

100 PRINT : PRINT CHR$(26) "THIS IS REVERSE MODE"
110 FOR I = 1 TO 10000: NEXT I    :  'DELAY LOOP
120 PRINT CHR$(25);:
130 PRINT "THIS IS NORMAL MODE"
```

When BASIC executes a program, it handles the statements one at a time, starting at the first and proceeding to the last. Some statements, such as GOTO, ON...GOTO, GOSUB, change this sequence.

STATEMENTS

A statement is a complete instruction to BASIC, telling the Computer to perform some operations. For example:

```
   GOTO 100
```

Tells the Computer to perform the operations of (1) locating line 100 and (2) executing the statement on that line.

```
   STOP
```

Tells the Computer to perform the operation of stopping execution of the program.

Many statements instruct the computer to perform operations with data. For example, in the statement:

```
   PRINT "SEPTEMBER REPORT"
```

the data is SEPTEMBER REPORT. The statement instructs the Computer to print the data inside the quotes.

EXPRESSIONS

An expression is actually a general term for data. There are two types of expressions:

1. Numeric expressions, which are composed of numeric data. Examples:

```
    (1 + 5.2) / 3                       D
    5 * B                               3.7682
    ABS(X) + RND(0)                     SIN(3 + E)
```

2. String expressions, which are composed of character data. Examples:

```
    A$                                  "STRING"
    "STRING" & "DATA"                    MO$ & "DATA"
    SEG$(A$,2,5) & SEG$("MAN",1,2)       M$ & A$ & B$
```

Functions

Functions are automatic subroutines. Most BASIC functions perform computations on data. Some serve a special purpose such as controlling the video display. You may use functions in the same manner that you use any data -- as part of a statement.

These are some of BASIC's functions:

```
    INT
    ABS
    STRING$
    SEG$
```

TESTS

BASIC will perform two kinds of tests to see if a certain kind of relationship exists between two or more expressions:

1. Relational tests, which test the equivalency relationship between the two expressions. Examples:

```
     A = 1
     A$ > B$
```

2. Logical tests, which test the logical relationship between relations. Examples:

```
      A$ = "YES" AND B$ = "NO"
      C > 5 OR M < B OR O > 2
```

For the rest of this chapter, we will cover in detail the way BASIC handles data and data operations, and how to input data into your program. The preceding overview should give you enough information if you are in a hurry to begin using Compiler BASIC.

HOW BASIC HANDLES DATA

This section provides information on how to represent data to BASIC and how BASIC will interpret and store it. It contains the necessary background information for writing programs which handle data efficiently.

WAYS OF REPRESENTING DATA

BASIC recognizes data in two forms -- either directly, as constants, or by reference to a memory location, as variables.

Constants

All data is input into a program as "constants" -- values which are not subject to change. For example, the statement:

```
   PRINT "1 PLUS 1 EQUALS"; 2
```

contains one string constant,

```
   1 PLUS 1 EQUALS
```

and one numeric constant

```
   2
```

In these examples, the constants are "input" to the PRINT statement. They tell PRINT what data to print on the Display.

These are more examples of constants:

```
   3.14159              "L. O. SMITH"
   1.775E+3             "0123456789ABCDEF"
   "NAME TITLE"         -123.45E-8
   57                   "AGE"
```

Variables

A variable is a place in memory -- a sort of box or pigeonhole -- where data is stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities. For example, in the statement:

```
   A$ = "OCCUPATION"
```

The variable A$ now contains the data OCCUPATION. However, if this statement appeared later in the program:

```
    A$ = "FINANCE"
```

The variable A$ would no longer contain OCCUPATION. It would contain the data FINANCE.

Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be upper or lower case and may be followed by up to 5 characters -- either digits or letters -- for a total of 6 characters.

For example

```
   AMOUNT      A      A12345      A1     B1AB2     aB
```

are all valid and distinct variable names.

Variable names may be longer than six characters. However, only the first six characters are significant in BASIC.

For example:

```
   SUPERN          SUPERNUM          SUPERNUMERARY
```

are all treated as the same variable by BASIC.

Reserved Words

BASIC has reserved certain words as BASIC functions. You cannot use these or the operator NOT as variable names. For example:

```
           ABS        SIN        LEN        ASC
```

cannot be used as variable names, because they are BASIC functions. However you can use reserved words inside variable names. For example, ABS1 and LENGTH are okay.

A BASIC statement may be used as long as it does not start the statement. For example:

```
     LET LET = 10
```

is okay, but

```
     LET = 10
```

is not.

Simple and Subscripted Variables

All of the variables mentioned above are simple variables (also termed scalars). They can only refer to one data item.

Variables may also be subscripted so that an entire list of data can be stored under one variable name. This method of data storage is called an array. For example, an array named A may contain these elements (subscripted variables):

```
   A(0)     A(1)     A(2)     A(3)    A(4)
```

You may use each of these elements to store a separate data item, such as:

```
   A(0) = 5.3
   A(1) = 7.2
   A(2) = 8.3
   A(3) = 6.8
```

```
A(4) = 3.7
```

In this example, array A is a one dimensional array, since each element contains only one subscript. An array may also be two dimensional, with each element containing two subscripts. For example, a two-dimensional array named X could contain these elements:

```
    X(0,0) = 8.6             X(0,1) = 3.5
    X(1,0) = 7.3             X(1,1) = 32.6
```

Compiler BASIC does not allow for more than two dimensions to an array.

Arrays must always be dimensioned before they are used, to reserve room in memory for them. The DIM statement dimensions arrays. Array A, in the example above would be dimensioned with:

```
    DIM  A(4)
```

to allow room for 5 subscripted variables (0, 1, 2, 3, and 4). Array X would be dimensioned with:

```
    DIM X(1,1)
```

to allow room for 2 subscripted variables in one dimension and 2 in the second dimension for a total of 2 * 2 = 4 subscripted variables.

Note: See DIM for more information on arrays.

HOW BASIC STORES DATA

The way that BASIC stores data determines the amount of memory it will consume and the speed in which BASIC can process that data.

Numeric Data

BASIC stores all numbers as either integer or real.

Integers
(Speed and Efficiency, Limited Range)

To be stored as an integer, a number must be whole and in the range of -32768 to 32767. An integer value requires only two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:

```
   1     32000     -2     500     -12345
```

can all be stored as integers.

NOTE: Integers are stored in two's complement notation. An explanation of that is in the Programmers Information Section.

Real Numbers
(Maximum Precision, Slower in Computations)

BASIC can store up to 14 significant digits when a number is stored as a real number. (it prints the first 6 digits, rounding off the last digit).

This is the range of real numbers:

```
   [-1 * 10 ^ -64, -1 * 10 ^ 63], or
   [1 * 10 ^ -64, 1 * 10 ^ 63]
```

A real number requires 8 bytes of storage. The first byte is for the exponent. Two digits of the number are stored in each of the next 7 bytes.

NOTE: An explanation of the way BASIC stores real numbers, in Binary Coded Decimal format, is in the Programmers Information Section.

String Data

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, text, etc. You may store any ASCII characters as a string. (A list of ASCII characters is in the Appendix).

For example, the data constant:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage. BASIC would store the above string constant internally as:

Hex	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38	Code
ASCII Char-acter	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8	

A string can be up to 255 characters long. Strings with length zero are called "null" or "empty".

HOW BASIC CLASSIFIES CONSTANTS

When BASIC encounters a data constant in a statement, it must determine the type of the constant (string, integer, or real). These are the rules it uses:

Rule 1

If the value is enclosed in double-quotes, it is a string. For example:

```
      "YES"
      "3331 Waverly Way"
      "1234567890"
```

the values in quotes are automatically classified as strings.

Rule 2

If the value has a & mark in front of it, it is a hexadecimal number. For example:

```
      &O          &7FCO          &FFFF
```

are all hexadecimal numbers. Hexadecimal numbers are actually stored as integers. You may use hexadecimal numbers in special cases such as in the EXT statement.

Rule 3

If the value is not in quotes, it is a number. (An exception to this rule is during data input by an operator. See INPUT, LINE INPUT, INKEY$, and INPUT$.)

For example:

```
       123001
       1
       -7.3214E+6
```

are all numeric data.

Rule 4

Whole numbers in the range of -32768 to 32767 are integers. For example:

```
        12350
        -12
        10012
```

are integer constants.

Rule 5

If the number contains a decimal point or is outside the integer range defined in rule 3 above, it is real. Also, if it contains the letter E, it is real.

Note: Exponents are printed with the letter E. The E indicates that the value printed multiplied by the specified power of 10 represents the data stored. For example:

```
      1. E+7
```

Represents the value 10000000, or 1 * 10 ^ 7.

```
       1. E-8
```

Represents the value .00000001 or 1 * 10 ^ -8.

HOW BASIC CLASSIFIES VARIABLES

When BASIC encounters a variable name in the program, it classifies it as either a string, integer or real number. It will only classify the variable name once in the program. You cannot get BASIC to re-classify a particular variable name.

These are the rules BASIC uses to classify variables:

Rule 1

Unless BASIC encounters a definition statement (described in rule 2 below) or a type declaration tag (described in rule 3 below), BASIC classifies all variable names as real number types and stores them in 8 bytes. For example:

```
    AB          AMOUNT          XY          L
```

are all real number variables initially. If this is the first line of your program:

```
     LP = 1.2
```

BASIC will classify LP as a real number variable.

Rule 2

If BASIC encounters a definition statement, BASIC will classify variables according to the instructions of that statement. There are three definition statements:

```
     STRING
     INTEGER
     REAL
```

The STRING Statement

STRING instructs BASIC to classify all variable names as string. For example:

```
     STRING
```

instructs BASIC to classify all variable names as string.

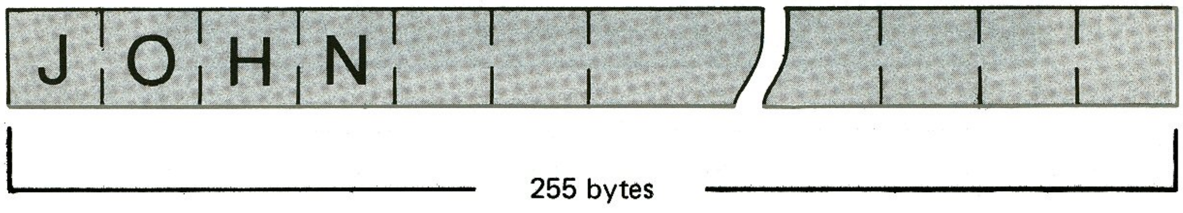
```
     STRING L
```

instructs BASIC to classify only those variable names beginning with the letter L as string.

BASIC assumes that all string variables should be stored in 255 bytes. For example, even though this statement only assigns 4 bytes of data to L:

```
     L = "JOHN"
```

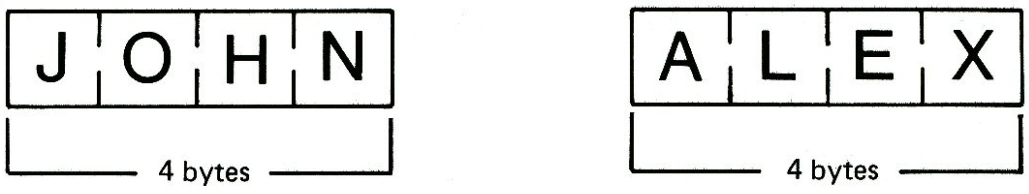
BASIC stores this data in 255 bytes. This causes L to contain 251 bytes of unused space.



To keep from wasting space in memory, you may specify the number of bytes to use in storing variables. For example, in this program:

```
     10 STRING*4 L
     20 L = JOHN
     30 LAST = ALEXANDER
```

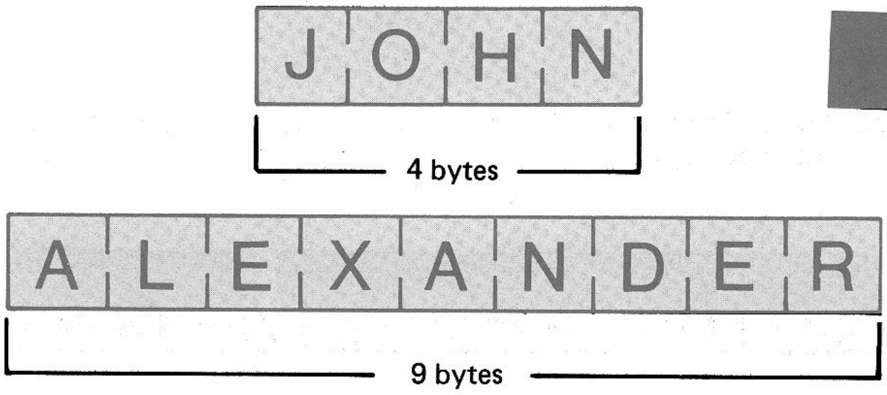
L and LAST will each contain 4 bytes of string data:



If you want to store all variable names beginning with the letter L as string variables except for the variable LAST, you can use the DIM statement:

```
      10 STRING*4 L
      20 DIM LAST$9
      30 L = JOHN
      40 LAST = ALEXANDER
```

This program stores the variable L in 4 bytes and LAST in 9 bytes.



Note: See DIM and STRING for more information.

The INTEGER Statement

INTEGER instructs BASIC to classify all variable names as integer. For example:

```
      INTEGER A
```

instructs BASIC to classify all variable names beginning with the letter A as integers.

```
      INTEGER
```

instructs BASIC to classify all variable names as integers.

In the present form of BASIC, all integer variables are stored in 2 bytes.

The REAL Statement

REAL instructs BASIC to classify variable names in its letter list as real numbers. For example, this program:

```
      10  INTEGER
```

```
     20  REAL X-Z
```

instructs BASIC to classify all variable names, except for those beginning with X, Y, or Z, as integers. BASIC will classify variable names beginning with X, Y, and Z as real.

In the present form of BASIC, all real number variables are stored in eight bytes.

Illegal Use of Definition Statements

You cannot introduce a definition statement after an executable statement. An executable statement is a statement other than a definition statement. For example:

```
     10 L = 10
     20 STRING
```

produces an error, since STRING may not follow the executable statement L = 10. However,

```
     10 STRING
     20 L = 10
```

is correct.

Rule 3

If a variable name has a type declaration tag following it, BASIC will classify it as string or integer according to the attributes of that tag:

$	String
%	Integer
#	Real

(However, you cannot use tags to re-classify variable names which BASIC has already classified previously in the program.)

For example, if the variable names S, MON, FINANCE, and CHART have not yet been used in the program:

```
       S$          MON$          FINANCE$          CHART$
```

will all be classified as string variable names, regardless of what attributes have been assigned to the letters S, M, F, and C.

If the variable names I, LM, NUM, and COUNTER have not yet been used:

```
        I%            LM%          NUM%         COUNTER%
```

will all be classified as integer variable names, regardless of what attributes have been assigned to the letters I, L, N, and C.

If the variables, LR, ER, MP235, and LITE have not yet been used:

```
      LR#          ER#          MP235#        LITE#
```

will all be classified as real number variables, regardless of what attributes have been assigned to the letters L, E, and M.

For example, in the program:

```
     10  STRING A
     20  AB = "NEW"
```

The statement:

```
     30  AB% = 1
```

produces an error, since AB has already been classified as a string variable and cannot be re-classified. However:

```
     30  AR% = 1
```

is accepted, since the type declaration tag (%) overrides the STRING A statement.

Once you use a type declaration tag to classify variables, you do not need to use the tag any more in the program. For instance, after this statement is executed:

```
      B$ = "DATA"
```

You may refer to the string variable B$ as simply B. B will retain the classification of a string variable throughout the rest of the program.

(Even though you only need to use the tag when you introduce the variable name, we suggest you use the tag every time you use the name. This makes the program more consistent and simplifies editing.)

HOW BASIC CONVERTS NUMERIC DATA

Often your program might ask BASIC to assign an integer data constant to a real number variable, such as:

```
     A = 5
```

or a real number constant to an integer variable, such as:

```
     B% = 5.2
```

To do this, BASIC must first convert the data constant. This is how it is done:

Real Number to Integer Type

BASIC truncates (ignores) the fractional part of the original value. The truncated value must be in the range of [-32768, 32767].

Examples

```
     A% = -10.5
```

Assigns A% the value -10.

```
     A% = 32767.9
```

Assigns A% the value 32767.

```
     A% = 2.5E+3
```

Assigns A% the value 2500

```
     A% = -123.45678901234
```

Assigns A% the value -123.

```
     A% = 60000
```

Prints an integer overflow warning and assigns A% the value 32767. (32767 is the highest number that can be stored as an integer).

Integer to Real Number Type

In converting integers to real numbers, the converted value is equal to the original value, but it consumes 4 times as much storage space. (Integers are stored in 2 bytes and real numbers in 8 bytes). For example:

```
     A = 1
```

Stores 1.0000000000000 in A.

Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
     A$ = 1234
     A% = "1234"
```

are illegal. (Use STR$ and VAL to accomplish such conversions).

HOW BASIC PERFORMS OPERATIONS ON DATA

This section explains how you can instruct BASIC to manipulate or test your data. The two means you have available are operators and functions.

OPERATORS

An operator is a single symbol or word which signifies some action to be taken on one or two specified values referred to as operands.

In general, an operator is used like this:

```
    operand-1 operator operand-2
        operand-1 and -2 can be expressions.  A few
          operations take only one operand, and are
          used like this:

    operator operand
         This is the form for a unary operation.
```

Examples:

```
     6 + 2
```

The addition operator + connects or relates its two operands, 6 and 2, to produce the result 8.

```
     -5
```

The negation operator - acts on a single operand 5 to produce the result negative 5.

Neither 6 + 2 or -5 can stand alone; they must be used in statements to be meaningful to BASIC. For example:

```
     A = 6 + 2
     PRINT -5
```

Operators fall into three categories:

Numeric
String
Test

based on the kinds of operands they require and the results they produce.

Numeric Operators

Numeric Operators are used in numeric expressions. Their operands must always be numeric, and the result they produce is one numeric data item.

In the descriptions below, we use the terms integer and real operations. Integer operations involve two-byte operands, and real operations involve eight-byte operands. Real operations are slower, since they involve more bytes.

There are nine different numeric operators. Two of them, sign + and sign -, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
     PRINT -77, +77
```

the sign operators - and + produce the values negative 77 and positive 77, respectively.

Note: When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands. These operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Integer division (keyboard character <CTRL 9 >
** or ^	Exponentiation (keyboard character <SHIFT 6>
MOD	Modulus arithmetic

Addition

The + operator is the symbol for addition. If both operands are integers, BASIC will perform integer addition. Otherwise, BASIC will convert any operands that are integers to real numbers, and perform real number addition.

Note: See the section on How BASIC Converts Data (earlier in this chapter) for an explanation on how integers are converted to real numbers.

Examples:

```
     PRINT 2 + 3
```

Integer addition.

```
     PRINT 30000 + 10000
```

Integer addition. Since the upper limit for integers is 32767, BASIC prints an overflow error warning.

```
     PRINT 1.2 + 3
```

Real number addition. (The integer 3 is converted to a real number).

Subtraction

The - operator is the symbol for subtraction. As in addition, both operands must be integers to perform integer subtraction.

Examples:

```
     PRINT 33 - 11
```

Integer subtraction

```
    PRINT 12.345 - 11
```

Real number subtraction.

Multiplication

The * operator is the symbol for multiplication. Once again, both operands must be integers to perform integer multiplication.

Examples:

```
     PRINT 33 * 11
```

Integer multiplication.

```
     PRINT 32000 * 10
```

Integer multiplication. Since the upper limit for integers is 32767, BASIC prints an overflow error warning.

```
     PRINT 12.345 * 11
```

Real number multiplication.

Division

The / symbol indicates ordinary division. Division is always with real numbers. If an operand is an integer, BASIC will convert it to a real number to perform real number division.

Examples:

```
     PRINT 3/4
```

Real number division.

```
     PRINT 3 / 1.2
```

Real number division.

Integer Division

The integer division operator \ is input by pressing <CTRL 9>. It converts its operands into integer type, then performs integer division. In integer division, the remainder is ignored, leaving an integer result. (If either operand is outside the range [-32768,32767], an error will occur.)

For example:

```
     PRINT 7 \ 3
```

prints the value 2, since 7 divided by 3 equals 2 remainder 1.

```
     PRINT -7 \ 3
```

prints -2.

Exponentiation

The symbol ^ (entered by pressing <SHIFT 6>) denotes exponentiation. It converts both its operands to real numbers and returns a real number result.

For example:

```
     PRINT 6 ^ .3
```

prints 6 to the .3 power.

Note: "**" may be used instead of "^".

Modulus Arithmetic

The MOD ("modulo") operator allows you to do modulus arithmetic. In modulus arithmetic, every number is converted to its equivalent in a cyclical counting scheme. For example, a 24-hour clock indicates the hour in modulo 24. Although the hour keeps incrementing, it is always expressed as a number from 0 to 23.

MOD requires two operands, for example:

```
     A MOD B
```

B is the modulus (the counting base) and A is the number to be converted.

(Expressed in mathematical terms, A MOD B returns the remainder after whole-number division of A by B. In the sense, it is the converse of \, which returns the whole number quotient and ignores the remainder.)

MOD converts both operands to integer type before performing the operation. If either operand is outside the range [-32768,32767], an error will occur.

Examples:

```
     PRINT 155 MOD 15
```

Prints 5, since 155/15 gives a whole number quotient of 10 with remainder 5.

```
     PRINT 79 MOD 12
```

Prints 7, since 79/12 equals 6 with remainder 7.

```
     PRINT -79 MOD 12
```

Prints -7.

```
     10 PRINT "TYPE IN AN ANGLE IN DEGREES"
     20 INPUT A%
     30 PRINT A; "="; A \ 90; " * 90 +"; A MOD 90
```

Input a positive angle greater than 90. Line 20 expresses the angle as a multiple of 90 degrees plus a remainder.

String Operator

BASIC has a string operator (&) which allows you to concatenate (link) two strings into one. This operator should be used as part of a string expression. The operands are both strings and the resulting value is one piece of string data.

The & operator links the string on the right of the & sign to the string on the left. For example:

```
     PRINT "CATS " & "LOVE " & "MICE"
```

prints:

```
     CATS LOVE MICE
```

Since BASIC does not allow one string to be longer than 255 characters, you need to be careful that your resulting string is not too long.

Test operators

You may use test operators in IF...THEN statements to test a certain kind of relationship between two or more expressions. This allows you to build elaborate decision-making structures into your programs. You may test either string or numeric expressions.

Test operators will return one of two results: True or False. BASIC has two kinds of test operators: relational and logical. The relational operators are <, >, and =; the logical operators are AND, OR, XOR, and NOT.

Relational operators

Relational operators compare two numerical or two string expressions. It then reports whether the comparison you set up in your program is true or false.

Numerical comparisons

This is the meaning of the operators when you use them to compare numeric expressions:

<	Less than
>	Greater than
=	Equal to
<> or ><	Not equal to
=< or <=	Less than or equal to
=> or >=	Greater than or equal to

Examples of true relations:

```
     1 < 2
     2 <> 5
     2 <= 5
     2 <= 2
     5 > 2
     7 = 7
```

Relational operators may only be used in an IF...THEN statement. For example

```
     IF A = 1 THEN PRINT "CORRECT"
```

BASIC tests to see if A is equal to 1. If it is, BASIC prints the message.

```
     IF X > 100 THEN 500
```

If the relation is true; that is, if X is larger than 100, than control branches to line 500.

String Comparisons

The relational operators for string expressions are the same as above, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare their alphabetical sequence. This allows you to sort string data:

<	Precedes
>	Follows
=	Has the same precedence
>< or <>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

BASIC compares the string expressions on a character-by-character basis. When it finds a non-matching character, it checks to see which character has the lower ASCII code. The character with the lower ASCII code is the smaller (precedent) of the two strings.

Note: The appendix contains a listing of ASCII codes for each character.

Examples

```
     "A" < "B"
```

The ASCII code for A is decimal 65; for B it's 66.

```
     "CODE" < "COOL"
```

The ASCII code for O is 79; for D it's 68.

If while making the comparison, BASIC reaches the end of one string before finding non-matching characters, the shorter string is the precedent. For example:

```
     "TRAIL" < "TRAILER"
```

Leading and trailing blanks are significant. For example:

```
     " A" < "A "
```

ASCII for the space character is 32; for A it's 65.

```
     "Z-80" < "Z-80a"
```

The string on the left is four characters long; the string on the right is five.

As with the numerical comparisons, these string comparisons can only be used in IF...THEN statements. These are examples of how they might be used:

```
     IF A$ < B$ THEN 50
```

If string A$ alphabetically precedes string B$, then the program branches to line 50.

```
     IF R$ = "YES" THEN PRINT A$
```

If R$ equals YES then the message stored as A$ is printed.

Logical Operators

Logical operators make logical comparisons. Like relational operators, they can only be used in IF/THEN statements and will only return a result of true or false. Except for the NOT operator, you may only use logical operators to compare two or more relations. For example:

```
     IF A = 1  OR  C = 2  THEN PRINT X
```

The logical operator, OR, compares the two relations A=1 and C=2.

Logical operators do not perform bit manipulations. Use the functions AND, OR, and XOR for that purpose.

This is how to use the logical operators:

AND

If both relations are true, then AND returns a logical true. Otherwise, it returns a logical false. For example:

```
     IF A = B AND B < 0 THEN 100
```

OR

If either of the relations is true, or both are true, OR returns a logical true. Otherwise it returns a logical false. For example:

```
     IF GAME = OVER OR TIME >= LATE THEN 500
```

XOR ("Exclusive OR")

Only when ONE of the relations is true (but not both) does XOR return a logical true. Otherwise it returns a logical false. For example:

```
     IF A$ = "YES" XOR B$ = "YES" THEN PRINT "ONLY ONE YES"
```

NOT

NOT is a unary operator, which means it only acts on one operand. The operand, like all the ones above, is a relation. When the relation is true, NOT returns a logical false. When it is false, NOT returns a logical true. For example:

```
     IF NOT(A$ < "M") THEN PRINT A$; "DOES NOT PRECEDE M"
```

Hierarchy of Operators

When your expressions have multiple operators, BASIC performs the operations according to a well-defined hierachy, so that results are always predictable.

Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expressions inside the parentheses before evaluating the rest of the expression. For example, the expression:

```
   8 - (3-2)
```

is evaluated like this:

```
     3 - 2 = 1
        8 - 1 = 7
```

With nested parentheses, BASIC starts evaluating the innermost level first and works outward. For example:

```
     4 * (2 - (3 - 4))
```

is evaluated like this:

```
     3 - 4 = -1
         2 - (-1) = 3
             4 * 3 = 12
```

Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, BASIC uses a hierachy to determine what operation to do first.

The two listings below show the hierarchy BASIC uses. Operators are shown in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed as encountered FROM LEFT TO RIGHT.

Numerical operations:

```
     ^ or **
     +, -  (unary sign operations -- not addition or
subtraction)
     *, /
     \
     MOD
     +, -
     <, >, =, <=, >=, <>
     NOT
     AND
     OR
     XOR
```

String operations:

```
     &
     <, >, =, <=, >=, <>
     NOT
     AND
     OR
     XOR
```

For example, in the line:

```
     X * X + 5^2.8
```

BASIC will find the value of 5 to the 2.8 power. Next, it will multiply X * X, and finally add this value to the value of 5 to the 2.8. If you want BASIC to perform the indicated operations in a different order, you must add parentheses. For example:

```
     X * (X + 5^2.8)
```

or

```
     X * (X + 5)^2.8
```

Here's another example:

```
     IF X = 0 OR Y > 0 AND Z = 1 THEN 255
```

The relational operators = and > have the highest precedence, so BASIC performs them first, one after the next, from left to right. Then the logical operations are performed. AND has a higher precedence than OR, so BASIC performs the AND operation before OR.

If the above line looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
     IF X = 0 OR ((Y>0) AND (Z=1)) THEN 255
```

FUNCTIONS

A function is a built-in sequence of operations which BASIC will perform on data. A function is actually a subroutine which usually returns a data item. The BASIC Compiler's functions save you from having to write a BASIC routine, and they operate faster than a BASIC routine would.

A function consists of a keyword followed by the data that you specify. This data is always enclosed in parentheses and, if more than 1 data item is required, separated by commas.

If the data required is termed 'number' you may insert any numerical expression. If it is termed 'string' you may insert either a string constant or a string variable.

Examples:

```
     SQR(A + 6)
```

Tells BASIC to compute the square root of A + 6.

```
     SEG$(A$, 3, 2)
```

Tells BASIC to return a substring of the string A$, starting with the third character, with a length of 2.

Functions cannot stand alone in a BASIC program. Instead they are used in the same way you use expressions -- as the data in a statement.

For example

```
     A = SQR(7)
```

Assigns A the data returned as the square root of 7.

```
     PRINT SEG$(A$, 3, 2)
```

Prints the substring of A$ starting at the third character and two characters long.

If the function returns numeric data, it is a numeric function and may be used in a numeric expression. If it returns string data, it is a string function and may be used in a string expression.

SYNTAX OF EXPRESSIONS

Understanding the syntax of expressions will help you put together powerful statements -- instead of using many short ones.

As we have stated before, an expression is actually data. This is because once BASIC performs all the operations, it returns one data item. An expression may be either a string or numeric expression. It may be composed of:

Constants
Variables
Operators
Functions

Expressions may be either simple or complex:

A SIMPLE EXPRESSION consists of a single TERM: a constant, variable or function. If it is a numeric term, it may be preceded by an optional + or - sign.

For example:

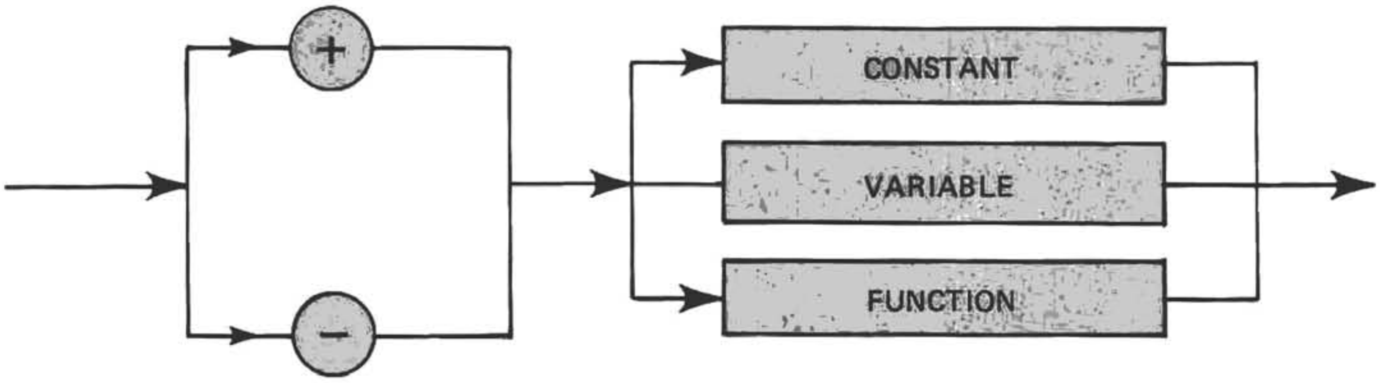
```
+A        3.3         -5          SQR(8)
```

are all simple numeric expressions, since they only consist of one numeric term.

```
A$        STRING$(20, A$)      "WORD"       "M"
```

are all simple string expressions since they only consist of one string term.

Here's how a simple expression or a term is formed:



A COMPLEX EXPRESSION consists of two or more terms (simple expressions) combined by operators. For example:

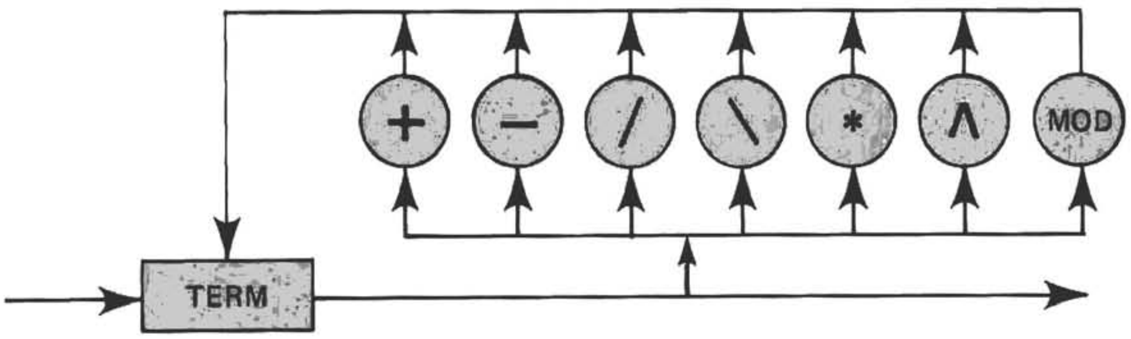
```
   A-1      X+3.2-Y      A/3 * (LOG(Y))   ABS(B) + LOG(2)
```

are all examples of complex numeric expressions.

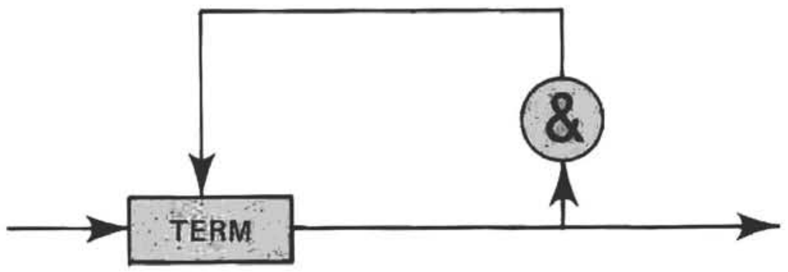
```
  A$ & B$     "Z" & Z$      STRING$(10, "A") & "M"
```

are all examples of complex string expressions.

This is how a complex numeric expression is formed:



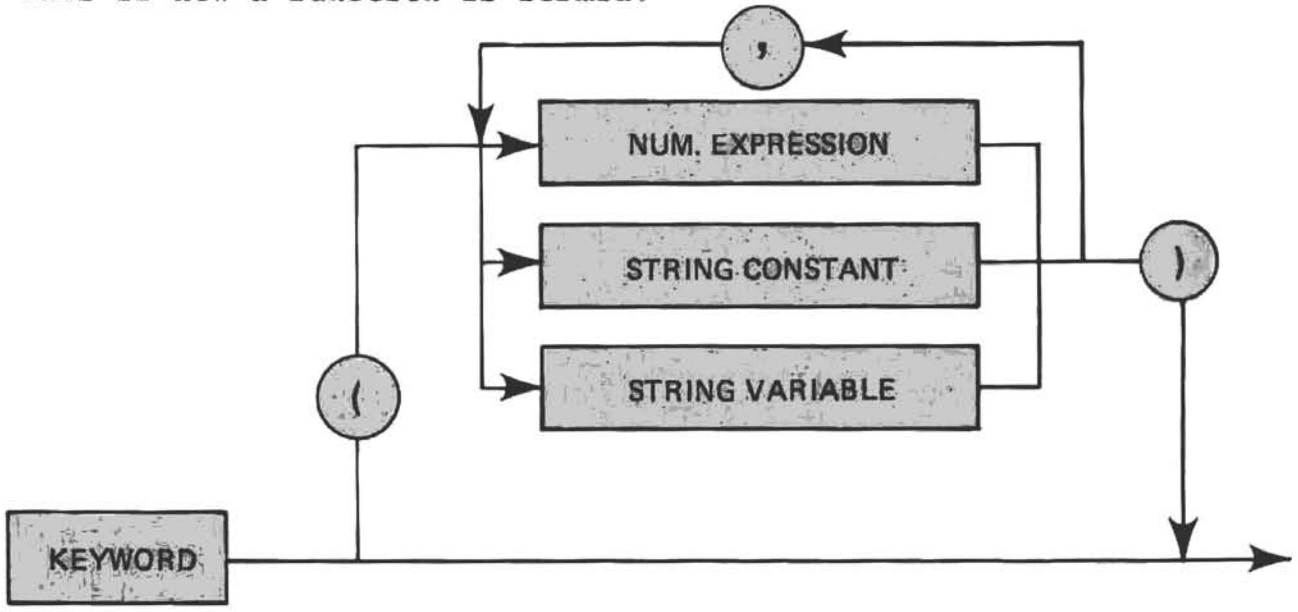
This is how a complex string expression is formed:



Most functions, except functions returning system information, require that you input either or both of the following kinds of data:

one or more numeric expressions
one or more string constants or string variables

This is how a function is formed:



If the data returned is a number, the function may be used as a term in a numeric expression. If the data is a string, the function may be used as a term in a string expresssion.

```
************************************************
*                                              *
*                Chapter 4                     *
*                                              *
*            BUILDING DATA FILES               *
*                                              *
************************************************
```

INTRODUCTION

This chapter explains how to write a BASIC program which will store data files on Model II diskettes. The Overview exPLains the different methods you can use to store data. The next sections run through the procedures to use in building the various types of data files.

OUTLINE FOR CHAPTER 4
BUILDING DATA FILES

OVERVIEW

INTRODUCTION TO DATA FILES

Data is stored on diskette in a data file. A data file is made up of records. Each record may contain from one to 256 bytes. Normally, one byte can hold one character of data.

For example, if the data file is a mailing list, each record could contain the data for one address. If the longest address contains 50 characters of data, the record would consume a little more than 50 bytes of space on the diskette.

A data file may contain as many records as you want and have room for. The system allocates space for each new record as you build the file. If you want to, you have the option of allocating space for your file in advance. To do this, use the TRSDOS "CREATE" command. (See the Model II Disk Operating System).

This overview covers:

1. the types of records you can build
2. the different ways you can access these records,
3. the methods you can use to input and output data to these records.

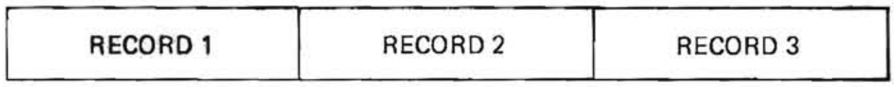
TYPES OF RECORDS

A data file may contain records which are fixed or varied in length:

Fixed Length Records (FLRs)

In a file containing FLRs, each record is the same length. This length can be from one to 256 bytes and is set the first time you open the file for use. Once set, the length may not be changed unless you are over-writing the file with new data.

This is a picture of an FLR file containing three records:

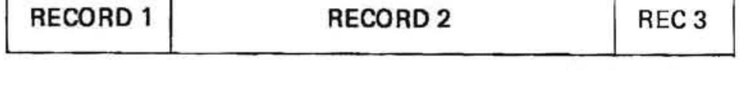


The advantage of using FLRs is that the position of each record can be easily calculated. For this reason, you can immediately access any record in the file. For instance, to access the contents of record 3, you do not have to read the contents of the first two records.

The disadvantages are obvious. FLRs often contain a lot of empty space. Also, the record length must be determined in advance.

Variable Length Records (VLRs)

In a file containing VLRs, each record may vary in length. Here is a picture of a VLR file containing three records:



Unlike FLRs, only the position of the first record and the end of the file can be located. To locate any other record, you must read each record in sequence, beginning with the first record, until you locate the record you want.

The advantage of using VLRs is that it is an easier and more flexible way of building a file. Virtually no space is wasted in a VLR file; each new record begins where the data in the last record ended.

WAYS OF ACCESSING RECORDS

There are three ways you may use to access a record in a file:

1. sequential access
2. direct access
3. indexed access

In sequential access, you must access each record sequentially. With direct access, you can access a record directly by referencing its record number. Indexed access allows you to access a record directly by referencing a key name which is indexed alphabetically.

Sequential Access

A sequential access file is normally made up of VLRs, although it may also be made up of FLRs. Since it is equipped for VLRs, only the first record and the end of the file can be directly accessed. Every other record must be accessed in sequence: record 1, record 2, record 3 ... the last record.

Using sequential access gives you the same advantages and disadvantages of using VLRs. It is a compact, easy, and flexible type of file to build , but it is time consuming to access individual records.

For instance, to update the file, you must read in every record, make any changes, and then write out each record to a new file on the diskette.

Some good uses for sequential access are:

1. Files which do not need to be accessed often, such as prior bookkeeping records.

2. Files which are only meant to be accessed in sequence, such as a file containing text information.

3. Files with widely varying record lengths.

4. Files where the maximum record length cannot be determined in advance.

Storage Format

In a variable length sequential access file, the first byte in each record gives the actual length of the record. This equals the amount of data plus one. Here is a picture of a record in a sequential access file:

7	R	E	C	O	R	D

In a fixed length sequential access file there is no count.

Direct Access

A direct access file (sometimes called random access) may only contain FLRs and has the advantages and disadvantages of FLRs. You assign each record a number when writing the record to the diskette. You may then use these record numbers to read or write to any record in the file.

Building a direct access file involves more planning than a sequential access file, since the record length must be determined in advance. To determine it, you need to calculate the maximum amount of data in a record, and how much space this record will consume on the diskette.

Some good uses for direct access are:

1. Files which contain standard sized records such as a mailing list.

2. Files which need to be continually updated such as inventory data.

Storage Format

This is a picture of a record in a direct access file which has a fixed length of 12 bytes of data for each record:

6	0	R	E	C	O	R	D						

The first byte of the record contains the actual number of bytes of data in the record. The second byte is not used in BASIC and is always the number 0.

The next bytes are for the actual data in the record. Since this record only has six bytes of data and the fixed record length has been set at 12 bytes, it contains six empty bytes.

Sometimes you might have a record containing no data in it, either because the record was deleted or no data was ever assigned to it. For example, say you had data in record 1 and record 3, but no data in record 2. Record 2 would still consume the same amount of space on disk as all the other records. This is what record 2 would look like:

0	0												

Often, after continually updating a direct access file, the file will contain a lot of deleted records and hence, a lot of empty space. To maintain this kind of file, you might periodically need to run a program which "packs" the data by assigning all the records new record numbers; thereby eliminating the space being consumed by deleted records.

Indexed Access (ISAM)

Like direct access, an indexed access file may only contain FLRs and offers the advantages and disadvantages of FLRs. Indexed files differ in the means of accessing the record. Rather that being accessed by a record number, the record is accessed by a key which you assign to the record when writing it to the diskette. This key may be any string whose length is the same as the length specified in the line which opens the file.

For example, each record in a payroll file could be assigned the person's last name as a key rather than a record number. This way you can use the person's last name, rather than looking up the record number, as a way of immediately accessing his or her record.

Indexed files are the easiest to operate and maintain. Operators can more easily use keys containing meaningful data than record numbers to access individual records in the file.

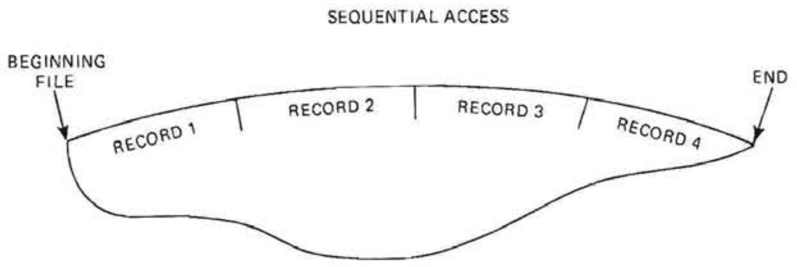
Maintaining an indexed file which has been updated frequently is also the easiest. Since a deleted record does not consume any space on the disk, it is not necessary to periodically run programs to pack all the records.

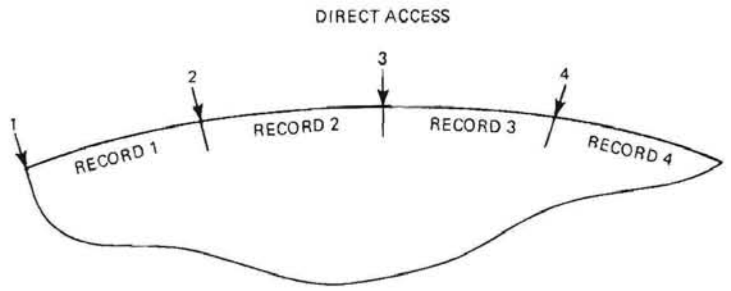
The disadvantage of indexed files is the amount of space they consume on the diskette. The overhead of the key index takes extra space. To build a file which uses disk space efficiently, you must carefully calculate the record length, key length, and number of records in the file. (The storage format is discussed in the Programmers Information Section).

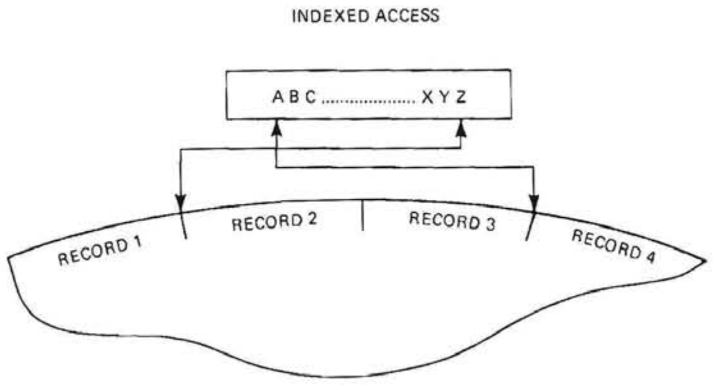
Some good uses for indexed access are:

1. Files which will be handled by many operators, such as checking account data at a bank.

2. Files which will continually have records inserted and deleted.


SEQUENTIAL ACCESS
BEGINNING FILE
END
RECORD 1
RECORD 2
RECORD 3
RECORD 4


DIRECT ACCESS
1
2
3
4
RECORD 1
RECORD 2
RECORD 3
RECORD 4


INDEXED ACCESS
A B C .... X Y Z
RECORD 1
RECORD 2
RECORD 3
RECORD 4

INPUT/OUTPUT METHODS

After deciding which type of records you will use and how to access the records, you need to decide how to input and output data to the records.

In choosing an input/output method, there are two things to consider:

1. how the data will be stored in the record
2. how the data will be fielded in the record

Fielding is a way of dividing data into different categories. For example, you might divide each record in a mailing list into five fields: (1) name, (2) address, (3) city, (4) state, (5) zip code. A record may contain as many data fields as you can fit in the record.

BASIC offers three methods of inputting and outputting data to a record:

1. Stream
2. Formatted
3. Binary

Each of these methods may be used with any type of records and with any type of access method.

The stream and formatted methods store each character of data in its ASCII format. This means each character consumes one byte of space on the diskette.

The binary method stores numeric data the same way it is stored in memory: integers in two bytes and real numbers in a maximum of nine bytes. For instance, the integer -23456 would consume six bytes of disk space with stream or formatted input/output, but only two bytes with binary.

The stream method separates each field by a comma. The formatted method formats the fields according to your specifications. The binary methods separates the fields by a length byte, or, if it is an integer, no field separator is necessary.

Note: In the following illustrations of stored records, only the data portion is shown. The beginning of the record would be in the format of the access method that is being used (sequential, direct, or indexed).

Stream Input/Output

When data is input and output in a stream, the PRINT statement outputs the data to the diskette, and the INPUT statement inputs data from the diskette. It is called the stream method because the length and format for the fields can differ with each record.

For example, if you were outputting records with three fields of data:

1. first name
2. last name
3. ID number

And this was the data for the first two records:

	First name (FIRST$)	Last name (LAST$)	ID (ID)
record 1	J	DAY	42
record 2	JANE	MILLER	2

You would input the data simply by using a comma to deliminate the end of one field and the beginning of the next field:

```
FIRST$, LAST$, ID
```

The data for these two records would be stored on the diskette in a stream with a comma separating each field

J	,	D	A	Y	,		4	2

J	A	N	E	,	M	I	L	L	E	R	,		2

Notice that each new field of data requires one extra byte of disk space for the comma.

Also note that a numeric field with a positive number requires one extra byte for a leading blank before the number. However if you output the ID as a string (ID$):

```
     FIRST$, LAST$, ID$
```

no leading blank would be required in storing the number:

J	,	D	A	Y	,	4	2

Stream input/output is best suited for VLRs, since the fields in each record may differ in length. However, the stream method may also be used with FLRs.

Formatted Input/Output

In formatted input/output, the INPUT USING and PRINT USING statements input and output data to the diskette. This allows you to use the image to control exactly how and where each field of data will be stored on the disk.

For example, you could output the same data as above using the formatted method with this image:

```
     <###<####<#
```

to format four characters for the first field, five for the second, and two for the third, with each field left justified. This is how the data would be stored:

J				D	A	Y			4	2

J	A	N	E	M	I	L	L	E	2	

Notice how each field of data is formatted to match the image line. Since the second field only allows for five left justified characters, the R in MILLER is truncated (deleted).

This is a good method to use when you need to be able to access any character of data in the record. For example, this method would make it easy to change the second character in each ID number.

Also, this is a good way to save disk space. If each field contains the same amount of data, the fields can be packed together in the record with no commas separating the them.

Binary Input/Output

In binary input/output, the READ and WRITE statements input and output data to the diskette.

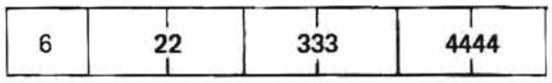
Numeric Data

Numeric data is stored much like it is in memory:

- integers are stored in two bytes, two's complement notation
- real numbers are stored in binary coded decimal format. This requires a maximum of nine bytes (the length byte plus the eight bytes for the number -- insignificant bytes are truncated.)

For an explanation of both of these storage formats, see the Programmers Information Section.

Integers must be whole numbers in the range of -32768 to 32767. For example, the integers 22, 333, 4444 would be stored as follows:



The first byte tells how may bytes of data are in the three following fields. Notice how each integer requires two bytes of storage. No extra bytes are required to separate each field.

The real numbers 2000 and 3333 would be stored in this format:

7	2	44	2	3	44	33	33
	FIELD 1 2000			FIELD 2 3333			

The field for the number 2000 consumes three bytes. The first byte, 2, tells the length of the field. The second byte, 44, is the exponent byte. The third byte, 2, contains the one significant digit in the number.

The next field for the number 3333 begins with the length byte, 3, which says that this field is four bytes long. The second byte, 44, is the exponent byte. The third and forth bytes contain the four significant digits in the number, 3333.

For more information on this, refer to the Programmers Information Section.

String Data

String data is stored in ASCII format with one byte per character plus a length byte to give the length of the string field.

The string data, "BINARY" and FILE" would be stored in a record in this form:

12	6	B	I	N	A	R	Y	4	F	I	L	E

Notice that each field contains a leading length byte.

Binary input/output is the most concise way to store a file containing largely numeric data. For example, a file containing sales data or accounting data would be best stored using the binary method.

BUILDING A SEQUENTIAL ACCESS FILE

As we discussed in the overview of this chapter, you have a choice of three methods you may use in building a sequential access file:

1. Stream method
2. Formatted method
3. Binary method

We will take you through the steps of building a sequential access data file using each of these methods. You will probably find it helpful, when going through these steps, to read about each statement we use. A write-up of each statement is in the Keywords Chapter of this manual.

SEQUENTIAL ACCESS USING STREAM INPUT/OUTPUT

The stream method is the most common way of building a sequential access file, since you do not have to format the length of the records in advance. We will show you how to use this method to:

1. build the file
2. read the file
3. add to the file

4. update the file

Building the File (Output to the File)

When building the file, you need to write a program that will do these four things:

1. Open the disk file with OPEN
2. Print a data record to the disk file with PRINT #
3. Repeat step 2 until your program has printed all the records to the disk file, and then
4. Close the file with CLOSE

Here is a sample program, along with a sample run of the program, which builds the file using these four steps:

```
    10 REM      *** DEMO OF STREAM OUTPUT TO A SEQUENTIAL FILE ***
    20 REM
    30 OPEN #1, "ITEM/DAT", MODE=W, TYPE=S
    40 PRINT "INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM"
    50 INPUT NO$, NAME$, DES$
    60 PRINT #1; NO$, NAME$, DES$
    70 PRINT "IS THERE ANOTHER ITEM (Y/N)?"
    80 INPUT ANSWER$
    90 IF ANSWER$ <> "N" THEN 40 ELSE CLOSE #1
 *RUN

 INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM
 ? 111
 ? PAPER
 ? LEGAL PAD 8 1/2 X 11 50 SHEETS
 IS THERE ANOTHER ITEM (Y/N)?
 ? Y
 INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM
 ? 222
 ? PEN
 ? BLUE INK BALL POINT MEDIUM INK
 IS THERE ANOTHER ITEM (Y/N)?
 ? N
```

Line 30 opens the file with the OPEN statement. (See OPEN):
- it references it as file unit #1. (You may have several

files open at the same time as demonstrated later in this section).
- it names it with the file specification of ITEM/DAT
- it sets the MODE to W since we are writing data to the file.
- it sets the TYPE to S for sequential access

Line 60 prints the data for one record to the file. This record has three fields: NO$, NAME$ and DES$. Notice that the PRINT # statement can only print one record to the disk file each time it is executed (See PRINT to a disk file).

Line 90 sets up a loop to continue printing as many records as you want to the disk file, and ...

When all the records are printed on the disk, line 90 closes the file.

Reading the File (Input from the File)

To read all the data records you have put in your file, you need to have your program do these five things:

1. Open the disk file with OPEN
2. Read in a data record with INPUT #
3. Use EOF to see if you have reached the end of the file yet.
4. Repeat steps 2 and 3 until you have read in all the records, and then
5. When you have reached the end of the file, close it with CLOSE

Here is a program, along with a sample run, which uses these steps to read in the file which was built above:

```
10 REM      *** DEMO OF STREAM INPUT FROM A SEQUENTIAL FILE ***
20 REM
30 OPEN #1, "ITEM/DAT", MODE=R, TYPE=S
40 INPUT #1; NO$, NAME$, DES$
50 IF EOF(#1) <> 0 THEN 90
60 PRINT : PRINT "ITEM NUMBER = ";NO$, "NAME = ";NAME$
70 PRINT "DESCRIPTION OF THE ITEM : "; DES$
80 GOTO 40
90 CLOSE #1
```

```
ITEM NUMBER = 111                   NAME = PAPER
DESCRIPTION OF THE ITEM : LEGAL PAD 8 1/2 X 11 50 SHEETS

ITEM NUMBER = 222                   NAME = PEN
DESCRIPTION OF THE ITEM : BLUE INK BALL POINT MEDIUM INK

STOP LINE 90
*SYSTEM "SCREEN"
```

Line 30 opens the file:
- again, it is file unit #1
- it names ITEM/DAT as the file to be opened (the file that was created above)
- it sets the MODE to R since we are reading data from the file
- it sets the TYPE, of course, as S for sequential

Line 40 causes your computer to INPUT (read) one data record from the disk file. It reads all three fields of the record. The first field is assigned to NO$, the second to NAME$, and the third to DES$.

Line 50 checks to see if you have reached the end of the file yet. If you have, it jumps to line 90 where the file is closed.

Line 40 sends the program back to INPUT or read another record, and

Line 90 closes the file.

Adding to the file

Should you decide at a later date that you want to add some more records to your file, you would follow a procedure almost identical to the one discussed above in "Building the File". The only difference is in the OPEN statement. Instead of setting the MODE to W (write), set it to E (extend).

Here is a sample program which extends the file built above named ITEM/DAT.

```
    10 REM      *** DEMO OF ADDING TO A SEQUENTIAL FILE ***
    20 REM
    30 OPEN #1, "ITEM/DAT", MODE=E, TYPE=S
    40 PRINT "INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM"
    50 INPUT NO$, NAME$, DES$
    60 PRINT #1; NO$, NAME$, DES$
    70 PRINT "IS THERE ANOTHER ITEM (Y/N)?"
    80 INPUT ANSWER$
    90 IF ANSWER$ <> "N" THEN 40 ELSE CLOSE #1
*RUN
INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM
? 333
? TYPEWRITER
? TAN ELECTRIC PORTABLE SELECTRIC
IS THERE ANOTHER ITEM (Y/N)?
? N
```

Updating the File

As we discussed in the overview of this chapter, updating a sequential access file is a time consuming process. These are the steps you need to follow:

1. Open the file you want to update (file #1) with OPEN
2. Open a second file with OPEN to write your updated records to (file #2)
3. Read in a data record with INPUT # from file #1
4. Use EOF to see if you have reached the end of file #1
5. Use PRINT # to print the updated record to file #2
6. Repeat steps 3, 4, and 5 until you reach the end of file #1, and then
7. Close file #1 with CLOSE
8. Kill file #1
9. Close file #2 with CLOSE

Here is a sample program which updates a sequential access file using these nine steps:

```
    10 REM      *** DEMO OF UPDATING A SEQUENTIAL FILE ***
    20 REM
    30 OPEN #1, "ITEM/DAT", MODE=R, TYPE=S
    40 OPEN #2, "NEWITEM/DAT", MODE=W, TYPE=S
    50 IF EOF(#1) = -1 THEN 160
    60 INPUT #1; NO$, NAME$, DES$
    70 PRINT : PRINT "ITEM NUMBER = ";NO$, "NAME = ";NAME$
```

```
 80 PRINT "DESCRIPTION OF THE ITEM : "; DES$
 90 PRINT : PRINT "DO YOU WANT TO CHANGE THIS INFORMATION (Y/N)";
100 INPUT ANSWER$
110 IF ANSWER$ = "N" THEN 140
120 PRINT "INPUT (1) ITEM NO$. (2) NAME (3) DESCRITPTION OF ITEM"
130 INPUT NO$, NAME$, DES$
140 PRINT #2; NO$, NAME$, DES$
150 GOTO 50
160 CLOSE #1
170 KILL "ITEM/DAT"
180 CLOSE #2
```

```
ITEM NUMBER =  111               NAME = PAPER
DESCRIPTION OF THE ITEM : LEGAL PAD 8 1/2 X 11 50 SHEETS

DO YOU WANT TO CHANGE THIS INFORMATION (Y/N)? N

ITEM NUMBER =  222               NAME = PEN
DESCRIPTION OF THE ITEM : BLUE INK BALL POINT MEDIUM INK

DO YOU WANT TO CHANGE THIS INFORMATION (Y/N)? Y
INPUT (1) ITEM NO. (2) NAME (3) DESCRITPTION OF ITEM
? 222
? PEN
? BLACK INK BALL POINT FINE LINE

ITEM NUMBER =  333               NAME = TYPEWRITER
DESCRIPTION OF THE ITEM : TAN ELECTRIC PORTABLE SELECTRIC

DO YOU WANT TO CHANGE THIS INFORMATION (Y/N)? N
```

Line 30 opens the file to be updated:
- it references it as file #1
- it names ITEM/DAT as the file to be opened
- it sets the MODE to R, since we will be reading data records from the file
- it sets the TYPE to S

Line 40 opens the second file which will contain the updated information:
- it references it as file #2
- it names this new file "NEWITEM/DAT"
- it sets the MODE to W, since we will be writing the updated data records to this file
- it sets the TYPE to S

Line 50 INPUTs (reads) one data record from file #1.

Line 60 checks to see if we have reached the end of file #1. If so, it sends program control to lines 160-180 where the two files are closed.

Line 140 PRINTS (writes) the updated record to file #2.

Line 150 sends the program back to read the next record, update it, and write the updated record to disk.

Line 160 closes file #1.

Line 170 kills file #1 since this file contains the old out-of-date information.

Line 180 closes the new file.

Notice that after running this program, you have created a new file named NEWITEM/DAT which contains your information.

SEQUENTIAL ACCESS USING FORMATTED INPUT/OUTPUT

Since the formatted method requires that you set the length of records in advance, it does not allow you to take advantage of the flexible record length that sequential access offers. However, you are still able to take advantage of the compactness of a sequential access file.

The steps for formatted input/output are identical to sequential input/output, except you need to replace PRINT # with PRINT USING # and INPUT # with INPUT USING #.

Sample programs:

```
    10 REM      *** DEMO OF FORMATTED OUTPUT TO A SEQUENTIAL FILE ***
    20 REM
    30 OPEN #1, "ITEM/DAT", MODE=W, TYPE=S
    40 PRINT "INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM"
    50 INPUT NO$, NAME$, DES$
    60 PRINT USING #1; 200, NO$, NAME$, DES$
    70 PRINT "IS THERE ANOTHER ITEM (Y/N)?"
    80 INPUT ANSWER$
    90 IF ANSWER$ <> "N" THEN 40 ELSE CLOSE #1
   200 ;<##<####<##############
*RUN
INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM
? 111
? PAPER
? LEGAL PAD 8 1/2 X 11 50 SHEETS
IS THERE ANOTHER ITEM (Y/N)?
? Y
INPUT (1) ITEM NO. (2) NAME (3) DESCRIPTION OF ITEM
? 222
? PEN
? BLUE INK BALL POINT MEDIUM POINT
IS THERE ANOTHER ITEM (Y/N)?
? N
```

```
   10 REM      *** DEMO OF FORMATTED INPUT FROM A SEQUENTIAL FILE ***
   20 REM
   30 OPEN #1, "ITEM/DAT", MODE=R, TYPE=S
   40 INPUT USING #1; 100, NO$, NAME$, DES$
   50 IF EOF(#1) <> 0 THEN 90
   60 PRINT : PRINT "ITEM NUMBER = ";NO, "NAME = ";NAME$
   70 PRINT "DESCRIPTION OF THE ITEM : "; DES$
   80 GOTO 40
   90 CLOSE #1
  100 ;<##<####<##############
*RUN


ITEM NUMBER = 111              NAME = PAPER
DESCRIPTION OF THE ITEM : LEGAL PAD 8 1/2

ITEM NUMBER = 222              NAME = PEN
DESCRIPTION OF THE ITEM : BLUE INK BALL P
```

SEQUENTIAL ACCESS USING BINARY INPUT/OUTPUT

To use the binary input/output method, use the same procedures as the stream input/output method, except replace PRINT # with WRITE and INPUT # with READ.

Sample Programs:

```
   10 REM      *** DEMO OF BINARY OUTPUT TO A SEQUENTIAL FILE ***
   20 REM
   30 OPEN #1, "SALES/DAT", MODE=W, TYPE=S
   40 PRINT "INPUT (1) ITEM NO. (2) JAN SALES (3) FEB SALES (4) MAR SALES"
   50 INPUT NO%, JAN, FEB, MAR
   60 WRITE #1; NO%, JAN, FEB, MAR
   70 PRINT "IS THERE ANOTHER ITEM (Y/N)";
   80 INPUT ANSWER$
   90 IF ANSWER$ <> "N" THEN 40 ELSE CLOSE #1
*RUN

INPUT (1) ITEM NO. (2) JAN SALES (3) FEB SALES (4) MAR SALES
? 111
? 1000
? 2000
? 3000
IS THERE ANOTHER ITEM (Y/N)? Y
INPUT (1) ITEM NO. (2) JAN SALES (3) FEB SALES (4) MAR SALES
? 222
? 1500
? 2000
? 2500
IS THERE ANOTHER ITEM (Y/N)? N
```

```
   10 REM      *** DEMO OF BINARY INPUT FROM A SEQUENTIAL FILE ***
   20 REM
   30 OPEN #1, "SALES/DAT", MODE=R, TYPE=S
   40 PRINT "ITEM NO", "JAN SALES", "FEB SALES", "MAR SALES"
   50 READ #1; NO%, JAN, FEB, MAR
   60 IF EOF(#1) <> 0 THEN 90
   70 PRINT NO%, JAN, FEB, MAR
   80 GOTO 50
   90 CLOSE #1
*RUN

ITEM NO             JAN SALES           FEB SALES           MAR SALES
 111                 1000                2000                3000
 222                 1500                2000                2500
```

BUILDING A DIRECT ACCESS FILE

As with sequential access, you may either use the stream, formatted, or binary methods to input and output data to a direct access file. We will discuss the formatted method first.

Again, in going through these sample programs, you will find it helpful to read about the keywords we use in the Keywords Chapter of this manual.

DIRECT ACCESS
USING FORMATTED INPUT/OUTPUT

Formatted input/output is a common way to build direct access files, since it will ensure that each record has the same length and is in the same format.

Building the file

Building a direct access file is actually very similar to the procedure of building a sequential file. The difference is:

- you must specify the length of each record in the OPEN statement
- you must assign each record a record number

These are the procedures to use:

1. Open the disk file with OPEN
2. Print a data record to the disk file with PRINT USING #, specifying its record number
3. Repeat step 2 until you your program has output all records desired to the disk file, and
4. Close the file with CLOSE

Here is a sample program following these procedures:

```
   10 REM       *** DEMO OF FORMATTED OUTPUT TO A DIRECT FILE ***
   20 REM
   30 OPEN #1, "LIST/DAT", MODE=W, TYPE=D, LENGTH=32
   40 X =1
   50 PRINT : INPUT PROMPT="LAST NAME ?"; LNAME$
   52 INPUT PROMPT="FIRST NAME ?"; FNAME$
   54 INPUT PROMPT="ADDRESS ?"; ADD$
   70 PRINT USING #1, KEY=X; 110, LNAME$, FNAME$, ADD$
   80 INPUT PROMPT="IS THERE ANOTHER ADDRESS (Y/N) ?"; ANSWER$
  100 IF ANSWER$ = "N" THEN CLOSE #1 ELSE X = X + 1 : GOTO 50
  110 ;<#########<######<##############
*RUN


LAST NAME ?HARRISON
FIRST NAME ?PATRICIA
ADDRESS ?1513 NORTH MOCKINGBIRD LANE
IS THERE ANOTHER ADDRESS (Y/N) ?Y

LAST NAME ?JOHNSON
FIRST NAME ?GEORGE
ADDRESS ?1811 SOUTH HAMPTON
IS THERE ANOTHER ADDRESS (Y/N) ?N
```

Line 110 is the image line. It determines how each record's data will be formatted on the diskette. In this program, each record will be divided into three fields. The < character marks the beginning of each field:

the first field has 10 characters;
the second, 7;
the third, 15.

for a total of 32 characters in each record.

Line 30 opens the file with OPEN:

- it references it as file unit #1
- it names the file "LIST/DAT
- it sets the MODE to W (write)
- it sets the TYPE to D (direct)
- it sets the LENGTH (record length) to 32 characters in each record.

Line 70 outputs a record to the disk file using the format set on line 110. Notice that in direct access, this PRINT USING # statement must specify a KEY (record number) for each record.

Line 100:
- closes the file if the operator does not want to output any more records, or
- increments the record number by 1 and sends the program back to print the next record to the disk file.

Reading the File (Input from the File)

To read every record in the file, you may use the same procedures that you would use in sequential access, except:

- in the OPEN statement, you must specify the record length
- in the INPUT USING # statement, you must specify the KEY (record number) you want to input from the file.

These are the procedures:

1. Open the disk file with OPEN, specifying the record length
2. Read in a data record with INPUT USING #, specifying the record number.
3. Use EOF to see if you have reached the end of the file yet.
4. Repeat steps 2 and 3 until you have read in all the records, and then
5. When you have reached the end of the file, close it with CLOSE.

Here is a sample program following these procedures:

```
10 REM      *** DEMO OF FORMATTED INPUT FROM A DIRECT FILE ***
20 REM
30 OPEN #1, "LIST/DAT", MODE=R, TYPE=D, LENGTH=32
40 X = 1
60 INPUT USING #1, KEY=X; 130, LNAME$, FNAME$, ADD$
65 IF EOF(#1) <> 0 THEN 120
```

```
    70 PRINT: PRINT "RECORD #"; X
    80 PRINT LNAME$;", ";FNAME$,,,,ADD$
   110 X = X + 1 : GOTO 60
   120 CLOSE #1
   130 ;<#########<######<##############
 *RUN

 RECORD # 1
 HARRISON   , PATRICI
 1513 NORTH MOCK

 RECORD # 2
 JOHNSON    , GEORGE
 1811 SOUTH HAMP
```

Line 130 is the image line determining what format to use in inputting each record from the disk file. This is the same image that was used in building the file.

Line 30 opens the file with OPEN:
- it references it as file unit #1
- it names it LIST/DAT
- it sets the MODE to R (read)
- it sets the TYPE to D (direct)
- it sets the LENGTH to 32 characters per record

Line 60 inputs record # X from disk, using the formatted image set in line 30. It assigns the three fields of data to the variables LNAME$, FNAME$, and ADD$.

Line 65 checks to see if you have reached the end of the file yet. If so, it jumps to line 120 where the file is closed.

Line 110 increments the record # by one and sends the program back to input the next record from disk.

Updating and Adding to the File

Direct access is the easiest way to update a file. Here are the procedures:

1. Open the file with OPEN, specifying the record length
2. By specifying the record number, you may then do one of the following:
   a. input the record from the disk file by using INPUT USING #
   b. delete the record from disk file with DELETE #, or
   c. output new data to the disk file, for

that record number with PRINT USING #
3. Repeat step 2 until you have finished updating the file, and then
4. Close the file with CLOSE

Here is a sample program updating a direct access file:

```
 10 REM      *** DEMO OF UPDATING A FORMATTED DIRECT FILE ***
 20 REM
 30 OPEN #1, "LIST/DAT", MODE=U, TYPE=D, LENGTH=32
 40 PRINT : PRINT "(1) DISPLAY RECORD" : PRINT "(2) DELETE RECORD"
 50 PRINT "(3) ADD/CHANGE" : PRINT "(4) CLOSE FILE"
 60 INPUT PROMPT="SELECT ONE OF THE ABOVE :"; S
 70 INPUT PROMPT="RECORD NO (Ø IF CLOSING FILE) ?"; R
 80 ON S GOTO 11Ø, 16Ø, 2ØØ, 27Ø
 90 REM
100 REM
110 REM      *** (1) DISPLAY RECORD ROUTINE ***
120 INPUT USING #1, KEY=R; 29Ø, LNAME$, FNAME$, ADD$
130 PRINT LNAME$;",";FNAME$,,,,ADD$ : GOTO 4Ø
140 REM
150 REM
160 REM      *** (2) DELETE RECORD ROUTINE ***
170 DELETE #1, KEY=R: GOTO 4Ø
180 REM
190 REM
200 REM      *** (3) ADD/CHANGE RECORD ROUTINE ***
210 INPUT PROMPT="LAST NAME ?"; LNAME$
220 INPUT PROMPT="FIRST NAME ?"; FNAME$
230 INPUT PROMPT="ADDRESS ?"; ADD$
240 PRINT USING #1,  KEY=R; 29Ø, LNAME$, FNAME$, ADD$ : GOTO 4Ø
250 REM
260 REM
270 REM      *** (4) CLOSE FILE ***
280 CLOSE #1
290 ;<#########<######<##############
```

Here is a sample of what might happen when this program is RUN:

```
*RUN

(1) DISPLAY RECORD
(2) DELETE RECORD
(3) ADD/CHANGE
(4) CLOSE FILE
```

```
SELECT ONE OF THE ABOVE :3
RECORD NO (Ø IF CLOSING FILE) ?3
LAST NAME ?ALEXANDER
FIRST NAME ?MARIA
ADDRESS ?3333 ELK GROVE


(1) DISPLAY RECORD
(2) DELETE RECORD
(3) ADD/CHANGE
(4) CLOSE FILE
SELECT ONE OF THE ABOVE :1
RECORD NO (Ø IF CLOSING FILE) ?3
ALEXANDER ,MARIA
3333 ELK GROVE

(1) DISPLAY RECORD
(2) DELETE RECORD
(3) ADD/CHANGE
(4) CLOSE FILE
SELECT ONE OF THE ABOVE :4
RECORD NO (Ø IF CLOSING FILE) ?Ø
```

Line 290 is the image line. This is format which was used when building the file.

Line 30 opens the file:
- it references it as file #1
- it names it LIST/DAT
- it sets the MODE to U (update)
- it sets the TYPE to D (direct)
- it sets the LENGTH to 32 characters per record

Line 70 asks the operator to input a record number (KEY)

Line 80 sends the program to the Display Routine, Delete Routine, Add/Change Routine, or to close the file, depending on the operators choice.

Line 120 inputs the record number the operator selected using the format set in line 290.

Line 170 deletes the record number the operator selected.

Line 240 prints new data to the record number the operator selected.

Line 280 closes the file.

DIRECT ACCESS
USING STREAM INPUT/OUTPUT

To use the stream input/output method, follow the procedures of the formatted method replacing PRINT USING # with PRINT # and INPUT USING # with INPUT #.

To determine the length of each record you must allot:
- one byte for each character of data.
- one byte for each comma which starts a new field of data.
- one byte preceding each positive number.
- one byte for the total length of the field.

Sample programs:

```
   10 REM      *** DEMO OF STREAM OUTPUT TO A DIRECT FILE ***
   20 REM
   30 OPEN #1, "NAME/DAT", MODE=W, TYPE=D, LENGTH=8
   40 X =1
   50 PRINT : PRINT "FIRST INITIAL ?";
   60 FNAME$ = INPUT$(1)
   70 PRINT "LAST NAME ?";
   80 LNAME$ = INPUT$(5)
   90 PRINT #1, KEY=X; FNAME$, LNAME$
  100 INPUT PROMPT="IS THERE ANOTHER NAME (Y/N) ?"; ANSWER$
  110 IF ANSWER$ = "N" THEN CLOSE #1 ELSE X = X + 1 : GOTO 50
*RUN

FIRST INITIAL ?M
LAST NAME ?WASHI
IS THERE ANOTHER NAME (Y/N) ?Y

FIRST INITIAL ?C
LAST NAME ?MILLE
IS THERE ANOTHER NAME (Y/N) ?Y

FIRST INITIAL ?J
LAST NAME ?SMITH
IS THERE ANOTHER NAME (Y/N) ?N
```

```
    10 REM      *** DEMO OF STREAM INPUT FROM A DIRECT FILE ***
    20 REM
    30 OPEN #1, "NAME/DAT", MODE=R, TYPE=D, LENGTH=8
    40 X = 1
    60 INPUT #1, KEY=X; FNAME$, LNAME$
    65 IF EOF(#1) <> 0 THEN 120
    70 PRINT: PRINT "RECORD #"; X
    80 PRINT FNAME$; ". "; LNAME$
   110 X = X + 1 : GOTO 60
   120 CLOSE #1

*RUN


RECORD # 1
M. WASH

RECORD # 2
C. MILL

RECORD # 3
J. SMIT
```

DIRECT ACCESS
USING BINARY INPUT/OUTPUT

To use the binary input/output method, follow the procedures of the formattted method replacing PRINT USING # with WRITE and INPUT USING # with READ.

Determining the length of each record is a little more complex. You should alot:

Bytes	
2	for each integer (integers are whole numbers beteen -32768 and 32767)
3 - 9	for each real number: 1 byte for the length byte 1 byte for the exponent byte 1 byte for each two signigicant digits
1	for the beginning length byte

See the Overview of this chapter for more information.

Sample programs:

```
 10 REM      *** DEMO OF BINARY OUTPUT TO A DIRECT FILE ***
 20 REM
 30 INTEGER
 40 OPEN #1, "SALES/DAT", MODE=W, TYPE=D, LENGTH=9
 50 X=1
 60 INPUT PROMPT = "ITEM NO. ?"; NO : INPUT PROMPT = "JAN SALES ?"; JAN
 70 INPUT PROMPT = "FEB SALES ?"; FEB : INPUT PROMPT = "MAR SALES ?"; MAR
 80 WRITE #1, KEY=X; NO, JAN, FEB, MAR
 90 PRINT "IS THERE ANOTHER ITEM (Y/N)";
100 INPUT ANSWER$
110 IF ANSWER = "N" THEN CLOSE #1 ELSE X = X + 1 : GOTO 60
```

```
*RUN
ITEM NO. ?111
JAN SALES ?3000
FEB SALES ?2433
MAR SALES ?5543
IS THERE ANOTHER ITEM (Y/N)? Y
ITEM NO. ?222
JAN SALES ?9987
FEB SALES ?8888
MAR SALES ?7987
IS THERE ANOTHER ITEM (Y/N)? N

STOP LINE 110
*SYSTEM "SCREEN"
```

```
 RSBASIC  ver 2.2          BI/DIR

     10 REM      *** DEMO OF BINARY INPUT FROM A DIRECT FILE ***
     20 REM
     30 INTEGER
     40 OPEN #1, "SALES/DAT", MODE=R, TYPE=D, LENGTH=9
     50 X=1
     60 PRINT "ITEM NO", "JAN SALES", "FEB SALES", "MAR SALES"
     70 READ #1, KEY=X; NO, JAN, FEB, MAR
     80 IF EOF(#1) <> 0 THEN 110
     90 PRINT NO, JAN, FEB, MAR
    100 X = X + 1 : GOTO 70
    110 CLOSE #1

*RUN

ITEM NO          JAN SALES        FEB SALES        MAR SALES
 111              3000             2433             5543
 222              9987             8888             7987
```

BUILDING AN INDEXED ACCESS (ISAM) FILE

To build an indexed access file, you may use the same three input/output methods that were shown with sequential and direct access files: formatted, stream, and binary. We will only show the formatted method in this chapter, but remember that the other methods are available to you.

INDEXED ACCESS FILE USING FORMATTED INPUT/OUTPUT

Building the File

To build the file, use the same procedures that were shown in building a formatted direct access file, except:

- In the OPEN statement, you must specify the maximum number of characters you will use for each KEY.
- In the PRINT USING # statement, you must assign each record a KEY rather than a record number. This key may be any name you choose but the length must eaual the key length you specified in the OPEN statement.

Here is a sample program:

```
 10 REM     *** DEMO OF FORMATTED OUTPUT TO AN INDEXED FILE ***
 20 REM
 30 OPEN #1, "LIST/DAT", MODE=W, TYPE=I, LENGTH=32, KEY=3
 40 PRINT : INPUT PROMPT="LAST NAME ?"; LNAME$
 50 INPUT PROMPT="FIRST NAME ?"; FNAME$
 60 INPUT PROMPT="ADDRESS ?"; ADD$
 70 PRINT "KEY (MUST BE EXACTLY 3 CHARACTERS) ? ";: K$=INPUT$(3)
 80 IF LEN(K$) < 3 THEN 70
 90 PRINT USING #1, KEY=K$; 120, LNAME$, FNAME$, ADD$
100 INPUT PROMPT="IS THERE ANOTHER ADDRESS (Y/N) ?"; ANSWER$
110 IF ANSWER$="N" THEN CLOSE #1 ELSE GOTO 40
120 ;<#########<######<##############
```

Line 120 is the image line. It formats the data output to each record in three fields containing 10, 7, and 15 characters for a total or 32 characters.

Line 30 opens the file:
- it references it as file unit #1.
- it names it LIST/DAT.
- it sets the MODE to W (write).
- it sets the TYPE to I (indexed).
- it sets the record LENGTH to 32.
- it sets the length of each KEY to 3 characters.

Line 70 asks the operator to specify a key name to use in referencing the file.

Line 80 is a trap to ensure that the length of K$ equals the key length set in the OPEN statement. If they are not equal, program execution returns to Line 70.

Line 90 prints the record to disk file.

Line 110 closes the file if the operator is finished or goes back to print another record to the disk file.

Reading the File

To read every record in the file, follow the same procedures that were shown in reading a formatted direct access file, except:

- In the OPEN statement, you must specify the number of charaters in the KEY>
- In the INPUT USING # statement, you may leave out the key name.
- You may use a special function named KEY$ to read the name of the key for each record.

Sample program:

```
 10 REM     *** DEMO OF FORMATTED INPUT FROM AN INDEXED FILE ***
 20 REM
 30 OPEN #1, "LIST/DAT", MODE=R, TYPE=I, LENGTH=32, KEY=3
 40 INPUT USING #1; 200,  LNAME$, FNAME$, ADD$
 50 IF EOF(#1) <> 0 THEN 100
 60 PRINT
 70 PRINT LNAME$;", "; FNAME$,,,ADD$
 80 PRINT KEY$
 90 GOTO 40
100 CLOSE #1
200 ;<#########<######<################
```

Updating the File

To update the file, you follow the same procedures as shown in updating a formatted direct access file, except:

- In the OPEN statement, you must specify the exact number of characters in the KEY.
- You must specify the name of the KEY in the INPUT USING #, PRINT USING # and DELETE # statements. (If you specify a KEY which is not in the data file or one which is not the correct length, then you COULD receive an "INVALID KEY" error and program execution would cease.)

```
 10 REM     *** DEMO OF UPDATING A FORMATTED INDEXED FILE ***
 20 REM
 30 OPEN #1, "LIST/DAT", MODE=U, TYPE=I, LENGTH=32, KEY=3
 40 PRINT : PRINT "(1) DISPLAY RECORD" : PRINT "(2) DELETE RECORD"
 50 PRINT "(3) ADD/CHANGE" : PRINT "(4) CLOSE FILE"
 60 INPUT PROMPT="SELECT ONE OF THE ABOVE :"; S
 70 INPUT PROMPT="KEY ?"; K$
 80 ON S GOTO 110, 160, 200, 270
 90 REM
100 REM
110 REM     *** (1) DISPLAY RECORD ROUTINE ***
120 INPUT USING #1, KEY=K$; 290, LNAME$, FNAME$, ADD$
130 PRINT LNAME$;",";FNAME$,,,,ADD$ : GOTO 40
140 REM
150 REM
160 REM     *** (2) DELETE RECORD ROUTINE ***
170 DELETE #1, KEY=K$: GOTO 40
180 REM
190 REM
200 REM     *** (3) ADD/CHANGE RECORD ROUTINE ***
210 INPUT PROMPT="LAST NAME ?"; LNAME$
220 INPUT PROMPT="FIRST NAME ?"; FNAME$
```

```
230 INPUT PROMPT="ADDRESS ?"; ADD$
240 PRINT USING #1,  KEY=K$; 290, LNAME$, FNAME$, ADD$ : GOTO 40
250 REM
260 REM
270 REM       *** (4) CLOSE FILE ***
280 CLOSE #1
290 ;<#########<######<##############
```

```
*******************************************
*                                         *
*               Chapter 5                 *
*                                         *
*          SEGMENTING PROGRAMS            *
*                                         *
*******************************************
```

WHY SEGMENT PROGRAMS

The BASIC Compiler offers two ways of segmenting long and complicated programs into shorter, more manageable programs:

1. Subprograms are high powered subroutines which act on data stored under different variable names. Like subroutines, they are called from the main program, executed, and return back to the main program.*

Subprograms are helpful if you are performing the same complicated operations on different variables repeatedly in different parts of your program. For example, a subprogram that draws graphs could be called many times from the program. Each time, it would be sent different data.

2. Program chaining is a method of breaking a very large program into smaller programs which will each load into memory and execute separately. This is a solution when a program requires too much memory to execute.

----

* A subprogram may also be called from another subprogram. However, they may not be recursive (that is, a subprogram may not call itself).

OUTLINE FOR CHAPTER 5
SEGMENTING PROGRAMS

HOW TO BUILD A SUBPROGRAM

All subprograms must be called from the main program with the CALL statement. Normally, you will want the CALL statement to "pass" data to the subprogram. For example:

```
     CALL "ANNUAL"; F
```

calls a subprogram named ANNUAL and passes the data stored in F to the subprogram.

The subprogram must begin with a SUB statement which identifies it. If the subprogram is being passed data, this statement must contain a variable name which can temporarily store the data. For example:

```
     SUB "ANNUAL"; X
```

begins the ANNUAL subprogram. The data in F is passed to the subprogram, which temporarily stores it as X. Here is the entire subprogram:

```
     100  SUB "ANNUAL"; X
     110  X = X * 52
     120  SUBEND
```

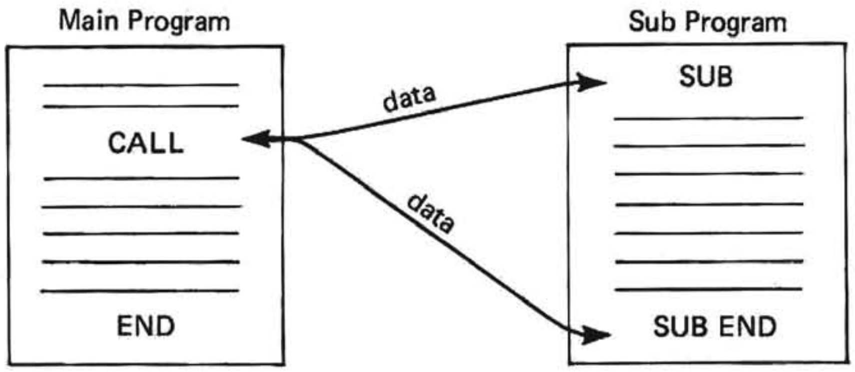
Notice that a subprogram must always end with a SUBEND statement. The main program must always end with an END statement. Here is the main program and the subprogram:

```
       5  X = 5
      10  F = 100
      20  CALL "ANNUAL"; F
      30  PRINT F
      40  END
     100  SUB "ANNUAL"; X
     110  X = X * 52
     120  SUBEND
```

Here, the main program passes the value of 100, which is stored in F, to the subprogram. The subprogram temporarily stores 100 in X, performs its operation on X and passes the resulting value of 5200 back to the variable F in the main program. When instructed to PRINT X and F, the main program prints:

```
     5           5200
```

Notice that the subprogram's variable X had no affect on the main programs's variable X. This is because subprogram and main program variables are stored separately. The subprogram only temporarily stores and acts on the value which is passed to it -- F.



The same subprogram may be called repeatedly in the program, being passed different values each time. For example:

```
 10  F = 100 : G = 52.25 : E = 26.50
 20  CALL "ANNUAL"; F
 30  CALL "ANNUAL"; G
 40  CALL "ANNUAL"; E
 50  PRINT F, G, E
 60  END
100  SUB "ANNUAL"; X
110  X = X * 52
120  SUBEND
```

When executed, this program prints:

```
5200                2717                1378
```

One CALL statement can pass several different variables to a subprogram. For example:

```
 10  MONTH$ = "JANUARY"
 30  DAY% = 5
 50  CALL "CAL"; MONTH$, DAY%
 60  PRINT MONTH$; DAY%
 90  SUB "CAL"; A$, B%
100  A$ = SEG$(A$, 1, 3)
110  B% = B% + 7
120  SUBEND
```

Notice that the variable types in the SUB statement (line 90) match the variables passed by the CALL statement (line 50). In this particular program, CALL and SUB list the string variable first and the integer variable second.

When executed, the program prints:

```
     JAN 12
```

Subprograms may be sent the contents of an entire array. For example:

```
     CALL "GRAPH"; A( )
```

calls the subprogram GRAPH and passes the entire contents of array A to the subprogram.

```
     SUB "GRAPH"; X( )
```

begins the subprogram GRAPH. The entire contents of array A are temporarily stored in the subprogram as array X.

Here is a program which passes array data to a subprogram:

```
  5  DIM A(3)
 10  DATA 5, 10, 15
 20  READ A(1), A(2), A(3)
 30  CALL "GRAPH"; A( ), "GRAPH"
 40  END
 50  SUB "GRAPH"; X( ), Y$
 60  PRINT Y$
 70  FOR I = 1 TO 3
 75  READ Z$: PRINT Z$;
 80  PRINT STRING$(X(I), "X"); X(I)
 90  NEXT I
 95  DATA "MON", "TUES", "WED"
100  SUBEND
```

Notice how the subprogram GRAPH beginning in line 50 has its own DATA statement (line 95). This cannot be read by the main program. Nor can the main program's DATA statement (line 5) be read by the subprogram. This is because before being executed, the main program and the subprogram are compiled separately.

You may pass the entire contents of a two dimension array like this:

```
     CALL "TWO"; A( , )
```

The subprogram needs a two dimensional array variable name to accept the contents of array A, such as:

```
     SUB "TWO"; X( , )
```

HOW TO PASS ALL TYPES OF DATA

The table on the next page shows how to match up the data in the CALL and SUB statement. The first column shows the type of data you may pass from the main program in a CALL statement. The second column shows the accompanying type of variable which must be in the SUB statement of the subprogram to receive this data.

DATA PASSED FROM THE MAIN PROGRAM	VARIABLE RECEIVER IN SUBPROGRAM
numeric expression CALL "SUBPROG"; 14 / 3 CALL "SUBPROG"; 14 * 3	numeric variable SUB "SUBPROG"; S SUB "SUBPROG"; S%
numeric variable contents CALL "SUBPROG"; M CALL "SUBPROG"; M%	numeric variable SUB "SUBPROG"; S SUB "SUBPROG"; S%
string constant contents CALL "SUBPROG"; "EXAMPLE"	string variable SUB "SUBPROG"; S$
string variable CALL "SUBPROG"; M$	string variable SUB "SUBPROG"; S$
entire one-dimensional numeric array contents CALL "SUBPROG"; M( ) CALL "SUBPROG"; M%( )	empty one-dimensional numeric array SUB "SUBPROG"; S( ) SUB "SUBPROG"; S%( )
entire two-dimensional numeric array contents CALL "SUBPROG"; M(,) CALL "SUBPROG"; M%(,)	empty two dimensional numeric array SUB "SUBPROG"; S(,) SUB "SUBPROG"; M%(,)
contents of numeric array element CALL "SUBPROG"; M(1) CALL "SUBPROG"; M(1,1)	numeric subscripted variable SUB "SUBPROG"; S SUB "SUBPROG"; S

CALL "SUBPROG"; M%(1) CALL "SUBPROG"; M%(1,1)	SUB "SUBPROG"; S% SUB "SUBPROG"; S%
entire one-dimensional string array contents CALL "SUBPROG"; M$( )	empty one-dimensional string array SUB "SUBPROG"; S$( )
entire two-dimensional string array contents CALL "SUBPROG"; M$(,)	empty two-dimensional string array SUB "SUBPROG"; S$(,)
contents of one string array element CALL "SUBPROG"; M$(1) CALL "SUBPROG"; M$(1,1)	string subscripted variable SUB "SUBPROG"; S$ SUB "SUBPROG"; S$

STORING SUBPROGRAMS

Subprograms may either be SAVEd or COMPILEd as part of the main program or as a separate program. If they are stored separately, they must be loaded along with the main program.

If the subprogram and main program were both SAVEd separately as BASIC programs, use the APPEND command to load the subprogram. For example:

```
     OLD MAINPRG/BAS
```

Loads the main BASIC program, and

```
     APPEND SUBPRG/BAS
```

Appends the subprogram to the main program.

CALLING ASSEMBLY LANGUAGE PROGRAMS

RSBASIC provides a method for calling an external assembled object code program from your BASIC program. To do this, use these guidelines:

When writing the assembly language program ...

1. We suggest that you calculate the originating address for your assembly language program as follows:

```
     TRSDOS TOP memory address*
   - number of bytes in your program
   ---------------------------------
     originating address
```

* your TRSDOS TOP memory address depends on the size of your system, which version of TRSDOS you have, and whether you will load high overlay programs such as DEBUG and SETCOM. The lowest possible TOP memory address you could have on a 64K system is F000.

2. If the subprogram will receive parameters passed to it by the main BASIC program, refer to the section on "Parameter Passing" of Assembly Language Subprograms in the Programmers Information Section. The sample program on the following pages demonstrates an application of how this is done beginning on line 4000 "CALL PARAMETER DECODING ROUTINE".

When writing the BASIC program...

1. Use the EXT statement to define this address and to name the subprogram. For example:

```
EXT DISKID = &A000
```

assigns the name DISKID to the subprogram and defines its originating address as hex A000.

The EXT statement should be at the beginning of your program.

2. Use the CALL statement to call the assembled program in the same manner that CALL is used to call a BASIC subprogram. For example:

```
CALL "DISKID"; DRIVE%, DISKID$
```

calls the subprogram named DISKID and passes the parameters (data) stored in DRIVE% and DISKID$.

When executing the program ...

1. Load RSBASIC specifying the top memory address it may use. This address should be the originating address of your assembled subprogram minus one. For example, if your originating address is A000, you should load RSBASIC with the T=9FFF option. (See Using the BASIC Compiler, Chapter 1 for the correct syntax).

2. After loading RSBASIC, you may load your assembled subprogram using the BASIC "SYSTEM" command and the TRSDOS "LOAD" command. For example:

```
SYSTEM "LOAD EX/OBJ:1"
```

loads the assembled program EX/OBJ from the diskette in drive 1.

```
1000 REM    DISKID/BAS
1010 REM
1020 REM    DEMONSTRATION OF A CALL TO AN EXTERNAL MACHINE-LANGUAGE (M-L)
1030 REM    SUBROUTINE. BEFORE RUNNING THIS PROGRAM, LOAD THE M-L
1040 REM    SUBROUTINE 'DISKID' INTO MEMORY. BASIC TOP OF MEMORY MUST
1050 REM    BE SET TO HEX ADDRESS 9FFF, E.G., START BASIC THIS WAY:
1060 REM      RMBASIC <T=9FFF>
1070 REM
1080 REM    THE M-L SUBROUTINE CHECKS FOR A VALID PARAMETER LIST AND
1090 REM    FOR VALID PARAMETERS
1100 REM
1110 DIM DISKID$8
1120 EXT DISKID = &A000
1130 PRINT "ENTER THE DRIVE NUMBER"
1140 INPUT DRIVE%
1150 CALL "DISKID"; DRIVE%, DISKID$
1160 LENGTH% = LEN(DISKID$)
1170 PRINT "LENGTH OF DISK NAME IS"; LENGTH%
1180 IF LENGTH% = 0 THEN 1200
1190 PRINT "THE DISK NAME IS "; DISKID$
1200 PRINT: GOTO 1130
```

```
01000   ;--------------------------------------------------------------------
01100   ;         DISKID -- EXTERNAL SUBROUTINE FOR RSBASIC
01200   ;
01300   ; 03/31/1980
01400   ;
01500   ; THIS EXTERNAL ROUTINE GETS THE DISKNAME AND RETURNS IT TO THE
01600   ; BASIC PROGRAM. ON ENTRY TO DISKID, REGISTER CONTENTS ARE:
01700   ;         (BC) = PARAMETER LIST
01800   ;         (DE) = SUBROUTINE TO RETURN NECESSARY PARAMETER INFORMATION
01900   ;                EACH TIME THIS SUBROUTINE IS CALLED, IT RETURNS THE
02000   ;                PERTINENT INFO ABOUT THE NEXT ARGUMENT IN THE LIST.
02100   ;
02200   ;--------------------------------------------------------------------
02300   ; INITIALIZATION SECTION
02400   ;
02500           ASEG
02600           ORG     0A000H
02700   SVCERR  EQU     52      ; FUNCTION CODE FOR ERRMSG-SVC
02800   SVCWRT  EQU     09      ; CODE FOR VDLINE-SVC
02900   SVCDID  EQU     15      ; CODE FOR DISKID-SVC
03000   ;
```

```
;----------------------------------------------------------------------
DISKID:                                ; ROUTINE STARTS HERE
        LD      HL,PDRADR              ; SAVE ADDR OF PARAMETER DECODING
        LD      (HL),E                 ; ROUTINE IN (PDRADR)
        INC     HL
        LD      (HL),D
; NOW (PDRADR) = POINTER TO PARAMETER-DECODING ROUTINE
;
;------------------------------------------------------------------------
; CALL PARAMETER DECODING ROUTINE
        LD      HL,CTNU1               ; SAVE CONTINUATION ADDRESS
        PUSH    HL                     ;   ON STACK
        LD      HL,(PDRADR)            ; CALL PDR
        JP      (HL)
;
; NOW A = RETURN CODE (0 => MORE ITEMS LEFT; 1 => NO MORE LEFT)
;    B = PARAMETER TYPE (0 => INTEGER, 1 => REAL, 2 => STRING)
; DE = ADDRESS OF ARGUMENT OR ARGUMENT DOPE
;
;------------------------------------------------------------------------
CTNU1:                                 ; EDIT FIRST PARAMETER
        CP      0                      ; 0 => MORE PARAMETERS IN LIST
        JR      NZ,PRMERR              ; ERROR IF THIS IS LAST PARAMETER
        CP      B                      ; B = PARAMETER TYPE (0=>INTEGER)
        JR      NZ,PRMERR              ; ERROR IF NOT INTEGER
;
; NOW CHECK FOR VALID PARAMETER (MUST BE 0, 1, 2, OR 3)
        INC     DE                     ; (DE) = MSB OF INTEGER
        LD      A,(DE)
        CP      0                      ; MSB SHOULD BE 0
        JR      NZ,DVERR               ; INVALID DRIVE NUMBER IF NOT 0
        DEC     DE                     ; (DE) = LSB OF INTEGER
        LD      A,(DE)
        CP      4                      ; SHOULD BE <= 3
        JR      NC,DVERR                       ; INVALID DRIVE NUMBER IF > 3
;
; NOW WE HAVE A GOOD DRIVE NUMBER IN REGISTER A
; SAVE IT IN 'DRIVE'
        LD      (DRIVE),A
;
;------------------------------------------------------------------------
; GET NEXT PARAMETER
        LD      HL,CTNU2               ; SAVE CONTINUATION ADDRESS
        PUSH    HL                     ;   ON STACK
        LD      HL,(PDRADR)            ; CALL PDR AGAIN
        JP      (HL)
;
```

```
;--------------------------------------------------------------------------------
; CHECK FOR VALID PARAMETER - SHOULD BE A STRING WITH LENGTH >= 8
CTNU2:  LD      A,2                 ; 2 => STRING TYPE
        CP      B
        JR      NZ,PRMERR           ; ERROR IF TYPE IS NOT STRING
;
; NOW (DE) = STRING DOPE: TEXT ADDRESS,MAXLENGTH
        LD      A,(DE)              ; LSB OF STRING DOPE
        LD      L,A                 ;   IS NOW IN L
        INC     DE                  ; NOW (DE) = MSB OF STRING DOPE
        LD      A,(DE)              ; GET IT INTO H
        LD      H,A                 ; NOW (HL) = STRING ADDR.: (LNTH,TEXT)
        LD      (BUFADR),HL         ; SAVE STRING ADDRESS IN (BUFADR)
        INC     DE                  ; NOW (DE) = MAXLEN
;
; NOW CHECK LENGTH OF STRING
        LD      A,(DE)              ; A = LENGTH-BYTE
        CP      8
        JR      C,BFERR             ; BUFFER SIZE ERROR ( < 8 )
        LD      (HL),A              ; ELSE SAVE LENGTH IN CRNT. LN. BYTE
;
;---------------------------------------------------------------------------------
; NOW READ THE DISKNAME
        LD      A,(DRIVE)
        LD      B,A
        LD      HL,(BUFADR)
        INC     HL                  ; SKIP CRNT. LN. BYTE
        LD      A,SVCDID
        RST     8                   ; DO SUPERVISOR CALL
        JR      NZ,SYSERR           ; HANDLE SYSTEM ERROR
        JR      ALLXIT              ; EXIT TO BASIC
;
;---------------------------------------------------------------------------------
; ERROR HANDLING ROUTINES
;
; HANDLE PARAMETER ERRORS
PRMERR:
        LD      HL,RMSG1            ; POINT TO ERROR MESSAGE
        LD      B,(HL)              ; B = LENGTH OF MESSAGE
        INC     HL                  ; (HL) = TEXT
        JR      SHOW                ; NOW DISPLAY THE MESSAGE
;
; HANDLE DRIVE NUMBER ERRORS
DVERR:
        LD      HL,RMSG2
        LD      B,(HL)
        INC     HL
        JR      SHOW
;
; HANDLE BUFFER SIZE ERRORS
BFERR:
```

```
12900           LD      HL,RMSG3
13000           LD      B,(HL)
13100           INC     HL
13200   ;
13300   ;---------------------------------------------------------------------
13400   ; DISPLAY ERROR MESSAGE (NON-SYSTEM ERRORS)
13500   SHOW:                           ; DISPLAY ERROR MESSAGE
13600           LD      C,20H           ; BLANK SPACE AFTER TEXT
13700           LD      A,SVCWRT        ; DO PRLINE-SVC
13800           RST     8
13900           LD      HL,RMSG4        ; POINT TO 'ERROR' LENGTH
14000           LD      B,(HL)          ; LENGTH
14100           INC     HL              ; TEXT AREA
14200           LD      C,0DH           ; CARRIAGE RETURN AFTER TEXT
14300           LD      A,SVCWRT        ; DO PRLINE-SVC
14400           RST     8
14500   ;
14600   ;       AFTER HANDLING NON-SYSTEM ERRORS, EXIT
14700           JR      ERRXIT
14800   ;
14900   ;---------------------------------------------------------------------
15000   ; HANDLE SYSTEM ERRORS
15100   SYSERR: LD      B,A             ; GET ERROR CODE
15200           LD      HL,RMSG5        ; (HL) = STORAGE AREA FOR MESSAGE
15300           LD      A,SVCERR        ; DO ERROR-SVC
15400           RST     8
15500           LD      B,80            ;LENGTH OF MESSAGE
15600           LD      C,0DH           ;CARRIAGE RETURN AFTER MESSAGE
15700           LD      A,SVCWRT        ; DO PRLINE-SVC
15800           RST     8
15900   ;
16000   ;---------------------------------------------------------------------
16100   ; AFTER ALL ERRORS, MUST ZERO CRNT STRING LENGTH
16200   ERRXIT: XOR     A
16300           LD      HL,(BUFADR)     ; (HL) = CRNT LN BYTE OF STRING
16400           LD      (HL),A          ; SET IT TO ZERO
16500   ;
16600   ;---------------------------------------------------------------------
16700   ALLXIT: RET                     ; RETURN TO BASIC PROGRAM
16800   ;
16900   ;
17000   ;---------------------------------------------------------------------
17100   ; DATA SECTION
17200   ;
17300   PDRADR: DW      00      ; WILL STORE PARAM. DECODING ADDRESS
17400   BUFADR: DW      00      ; WILL STORE POINTER TO DESTINATION TEXT
17500   DRIVE:  DB      0       ; WILL STORE DRIVE PARAMETER
17600   RMSG1:  DB      9       ; LENGTH OF MESSAGE
17700           DB      'PARAMETER'
17800   RMSG2:  DB      12
17900           DB      'DRIVE NUMBER'
18000   RMSG3:  DB      11
18100           DB      'BUFFER SIZE'
18200   RMSG4:  DB      5
18300           DB      'ERROR'
18400   RMSG5:  DS      80      ; STORAGE AREA FOR SYSTEM ERROR MESSAGE
18500           END DISKID
```

HOW TO CHAIN PROGRAMS

The CHAIN statement chains programs. For example:

```
     CHAIN "PROG2/BAS"
```

erases the program presently in memory, loads PROG2/BAS, and executes it.

```
     CHAIN "DRILL:2"
```

erases the program in memory and loads and executes DRILL from the disk in drive 2.

This is how program chaining could be used:

```
      10  PRINT "WHICH DRILLS TO YOU WANT TO TRY"
      20  PRINT "(1)ADDITION (2)SUBTRACTION (3)MULTIPLICATION
      30  INPUT X
      40  ON X GOTO 100, 200, 300
     100  CHAIN "ADD/CMP"
     200  CHAIN "SUBTR/CMP"
     300  CHAIN "MULT/CMP"
```

As with subprograms, you may pass data to the chained program. This is done with the COM statement. COM must be the first line in both the originating program and the chained program. For example this could be the originating program:

```
      10  COM A$
      20  PRINT "TYPE YOUR NAME"
      30  INPUT A$
      40
      50
      60
      70  CHAIN "TWO/BAS"
```

and the chained program could begin like this:

```
      10  COM A$
      20  PRINT "HELLO"; A$
      30  PRINT "THESE ARE THE FIRST 5 QUESTIONS"
```

Because of the COM A$ statement, the value of A$ is retained during the chaining process.

For more information on COM, see the Keywords Chapter.

SUBPROGRAMS VS. PROGRAM CHAINS

Subprograms are a good way to perform complicated routines on data repeatedly in the program, each time returning back to the main program. In chaining, it is more difficult to return back to the original program, since the main program is erased from memory when a program is chained.

Program chaining does offer a convenient way to write a program which requires more memory than there is available. The amount of memory you need to run a series of program chains is only the amount required to run the longest program in the series.

Subprograms do not have this memory saving capability. All subprograms must be loaded along with the main program prior to executing the program. There must be enough memory for the main program plus all the subprograms which will be called.

```
********************************************
*                                          *
*               Chapter 6                  *
*                                          *
*             BASIC KEYWORDS               *
*                                          *
********************************************
```

INTRODUCTION

The RSBASIC programming language is made up of keywords. These keywords, with their parameters, instruct the Computer to perform certain operations.

This chapter contains entries for each keyword, organized alphabetically. The first two pages show the meaning of the format for each keyword entry. A brief introduction to BASIC's two types of keywords -- statements and functions -- is on the next pages.

OUTLINE FOR CHAPTER 6
BASIC KEYWORDS

FORMAT FOR THE KEYWORD ENTRIES

A sample keyword entry is on the next page. This is the meaning of its format:

①. The first line is the keyword itself. The second line briefly describes what it does.

②. All keywords are defined as statements or functions:
    a. a STATEMENT is a line in a program. It, along with its parameters, tells the Computer to do some operation when that particular line in the program is executed.
    b. a FUNCTION is a subroutine. It must be a part of a statement.

③. The information in the gray box is the syntax for the keyword. The first line shows the format to use in typing the keyword. This format line always contains:
    a. the keyword itself - this must by typed exactly as it appears.
And may also contain:
    b. parameters
The parameters are defined on the next lines. A parameter enclosed in single quotes means that you must specify its value. Parameters may only be omitted if the syntax states that this is allowed.

In the syntax illustrated on the next page, LEN is the keyword and 'string' is the parameter. The second line gives the meaning of 'string'. Since 'string' is enclosed in single quotes, you must specify its value. The syntax does not state that 'string' may be omitted. Therefore 'string' is required.

④. This explains how to use the keyword.

⑤. These examples illustrate how the keyword might be used. All of these examples must be a line in the program to be executed.

⑥. Each entry contains a sample program using the keyword. Some of the longer sample programs illustrate a sample run of the program.

-- FUNCTION -- (2)

LEN
Get Length of String (1)

```
LEN(string)
  'string' is a string constant or a string variable.
```
(3)

LEN returns the current number of characters in the 'string'. (4)

Examples

```
   PRINT LEN("MARY")
```
(5)

Prints 4.

```
   PRINT LEN("MARY HAD A")
```

Prints 10.

```
   X = LEN(SENTENCE$)
```

Stores the number of characters in SENTENCE$ in X.

Sample Program

```
RSBASIC  ver 2.2          LEN/COM

   80 REM      *** SAMPLE PROGRAM DEMONSTRATING LEN ***
   90 REM
  100 PRINT "INPUT WORDS OR A SHORT SENTENCE"
  110 INPUT A$
  120 PRINT "YOUR SENTENCE HAS"; LEN(A$); "CHARACTERS"
  130 GOTO 100
```
(6)

STATEMENTS

A program is made up of lines; each containing one or more statements. A statement instructs the computer to do some operation when that particular line is executed. It may only be executed when the program is run. For example:

```
     100 STOP
```

Tells the Computer to stop executing the program when it reaches line 100.

Statements often include parameters. For example:

```
     100 GOTO 500
```

Tells the Computer, when it reaches line 100, to execute the statement on line 500 next.

BASIC statements perform the operations listed below:

VARIABLE DEFINITION

If none of the statements below are used, BASIC will treat all variables without a type declaration tag as real numbers, and no arrays will be allowed:

INTEGER - defines variables as integer
STRING - defines variables as string and defines the length of the string
REAL - defines variables as real
DIM - defines array variables, the length of array variables, and the length of string variables

The chapter on BASIC Concepts explains how BASIC handles variable definition.

ASSIGNING VALUES TO VARIABLES

BASIC allows you to assign values to variables directly or by using data statements:

DATA - stores data in your program so that you may assign

it to a variable.
LET - assigns a value to a variable (the keyword LET may be omitted)
READ - reads the data stored in the DATA statement and assigns it to a variable.
RESTORE - resores the pointer which points to a data item in the DATA statement.
SWAP - exchanges the values of variables

PROGRAM FLOW

The Computer will execute each line in the program sequentially, unless instructed to do otherwise. These statements change the flow of a program, either by branching within a program or segmenting a long program into shorter programs:

Branching within a Program

FOR/NEXT - establishes a program loop
GOSUB - transfers program control to the subroutine
GOTO - transfers program control to the specified line number
IF...THEN...ELSE - Performs the specified operation if the conditions are met.
ON...GOSUB - tests the value and branches to the subroutine
ON...GOTO - tests the value and branches to the program line specified.
RETURN - returns from the subroutine to the calling program
STOP - stops execution of the program

Segmenting Programs

CALL - transfers control to the subprogram
CHAIN - loads and executes the specified program
COM - stores variables in a common area so they may be passed to the chained program
EXT - defines the address of an external routine
END - ends compilation of main program
SUB - defines the beginning of the subprogram
SUBEND - returns execution back to the calling program

The chapter on Segmenting Programs explains how to segment programs.

INPUT/OUTPUT

Keyboard input statements allow the operator to input (type data into memory) from the keyboard. To print data, BASIC contains statements which output to the video display and line printer. Data is stored on disk by using input/output statements to a disk file.

Keyboard Input

INPUT - inputs data from the keyboard
INPUT USING - inputs formatted data from the keyboard
LINE INPUT - inputs a line of data from the keyboard

Output to the Display and Line Printer

LPRINT - prints data on the line printer
LPRINT USING - prints data on the line printer using the specified format
PRINT - prints data on the display
PRINT USING - prints data on the display using the specified format

Input/Output to a Disk File

CLOSE - closes a disk file
DELETE - deletes a record in a disk file
INPUT - inputs data from a disk file
INPUT USING - inputs data from a disk file using the specified format
KILL - kills a disk file
LINE INPUT - inputs a line of data from a disk file
OPEN - opens a disk file
PRINT - prints data to a disk file
PRINT USING - prints data to a disk file using the specified format
READ - reads binary data on a disk file
WRITE - writes binary data to a disk file

The chapter on Data Files explains how to use these statements.

DEBUGGING

These statements build an error trapping routine, which may be used in debugging a program or handling errors from a computer operator:

ERROR - simulates the specified error
ON BREAK GOTO - enables a <BREAK> handling routine
ON ERROR GOTO - enables an error trapping routine
RESET BREAK - disables the <BREAK> handling routine
RESET ERROR - disables the error trapping routine
RESET GOSUB - clears all the return addresses
RESUME - terminates the error handling routine

SPECIAL STATEMENTS

DEF - defines a function
RANDOMIZE - reseeds the random generator
REM - allows insertion of programmer's comment line
SYSTEM - returns the system to TRSDOS

FUNCTIONS

Functions are built-in subroutines. They may only be used as part of a statement.

Most BASIC functions perform certain routines to return numeric or string data. Special print functions are used to control the video display.

NUMERIC FUNCTIONS

All numeric functions return a number and may be used in a statement as numeric data. For example, the function:

```
SQR(9)
```

returns the number 3 (the square root of 9). This function may be used in a statement as numeric data. For example:

```
X = SQR(9)
```

assigns the square root of 9 to X.

Numeric functions perform these operations:

Arithmetic Operations

ABS	computes the absolute value
SGN	computes the sign (positive, negative, zero)
SQR	computes the square root

Converting Data to a Different Data Type

CVD	converts integer data to a real number
CVI	converts real data to an integer
HVL	converts a hexadecimal string to an integer
INT	converts real data to a whole number
VAL	converts numeric characters in a string to a number

Computations on Strings

ASC - returns the ASCII code of a string character
DIG - computes the length of numeric field in a string
LEN - computes the length of a string
POS - searches for a substring within a string

Bit Manipulation

AND - calculates the logical AND
OR - calculates the logical OR
XOR - calculates the exclusive XOR

Trigonometric Calculations

ATN - computes the arctangent
COS - computes the cosine
EXP - computes the natural exponential
EXP10 - computes the base 10 exponential
LOG - computes the natural logarithm
LOG10 - computes the base 10 logarithm
SIN - computes the sine
TAN - computes the tangent

Special System Information

CRTX - returns the row position of the cursor
CRTY - returns the column position of the cursor
ERR - returns the error code
EOF - notifies if the end of a disk file is reached
RND - returns a pseudo-random number

STRING FUNCTIONS

All string functions return a string and may be used in a statement as string data. For example, the function:

```
STRING$(5,"*")
```

returns the string ***** (5 asterisks). This function may be used in a statement as string data. For example:

```
     A$ = STRING$(5,"*")
```

assigns ***** to A$.

String functions perform these operations:

Converting Numbers to String

     CHR$ - returns the one-character string of the ASCII code
     HEX$ - converts an integer to a hexadecimal string
     STR$ - converts numeric data to string

Inputting a String

     INKEY$ - gets a keyboard character, if it has been pressed
     INPUT$ - inputs a character string from the keyboard

Manipulating a String

     SEG$ - returns a segment of a string
     STRING$ - returns a string of characters

Special System Information

     DATE$ - returns the date which was set when initializing the system
     TIME$ - returns the time recorded in the system's clock
     CRTI$ - returns the characters from a specified position on the video display

SPECIAL PRINT FUNCTIONS

Unlike numeric and string functions, the special print functions do not return data. Instead, they are used to control the video display. For example:

```
     CRT(5,7)
```

Moves the cursor to the row 5, column 7 position on the video display. This function may only be used in a PRINT statement. For example:

```
     PRINT CRT(5,7);"HEADING"
```

Prints HEADING at the row 5, column 7 position on the video display.

These are the special print functions:

CRT - moves the cursor to a specified row and column position
CRTR - moves the cursor relative to its current row and column position
CRTG - moves the cursor to a specified position and prints a string in the graphics mode
TAB - tabs the cursor to a specified column position

-- FUNCTION --

ABS
Compute Absolute Value

```
ABS(number)
   'number' is any numeric expression
```

ABS returns the absolute value of the 'number'. The absolute value is the magnitude of the number without respect to its sign.

ABS returns the same type of value (integer or real) as number.

Examples

```
PRINT ABS(3)
```

Prints 3.

```
   PRINT ABS(-3)
```

Prints 3.

```
   PRINT ABS(0)
```

Prints 0.

```
   X = ABS(Y + 3X)
```

The absolute value of Y + 3X is assigned to X.

```
   IF ABS(X) < 1E-6 THEN PRINT "TOO SMALL"
```

TOO SMALL is printed only if the absolute value of X is less than the indicated number.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING ABS ***
 90 REM
100 INTEGER A - Z
110 PRINT CHR$(27)
120 PRINT "GUESS MY NUMBER ";
130 X = RND(0) * 20 + 1
140 INPUT Y: IF X = Y THEN 170
150 PRINT "OFF BY"; ABS(X-Y); ".  GUESS AGAIN";
160 GOTO 140
170 PRINT "RIGHT!  GUESS MY NEXT NUMBER";
180 GOTO 130
```

-- FUNCTION --

AND
Calculate Logical AND

```
AND (number, number)
   'number' is any number in the range of
      -32768 to 32767.
```

AND is a logical operation performed on the binary representations of the two 'numbers'. AND compares each bit of the two numbers. A binary 1 is returned if both bits are a 1; a 0 is returned in any other case:

First Number	Second Number	Bit Returned
1	1	1
1	0	0
0	1	0
0	0	0

If 'number' is real, AND will convert it to an integer. The binary number that AND returns is always expressed as an integer.

Note: See also OR and XOR.

Examples

```
   PRINT AND (51, 15)
```

Prints a 3. The operation is performed on the binary representation of the two arguments:

```
                         Binary
    Integer           Representation
      51                 00110011
      15                 00001111
     -----              ----------
       3                 00000011
    A = AND (51, 3)
```

Performs AND operation and assigns the value of 3 to A.

The two examples below illustrate a common use of AND. All other bits can be masked out to see if one particular bit is "on" (1):

```
    IF AND (128, 64) = 64 PRINT "TRUE" ELSE PRINT "FALSE"
```

Prints "FALSE".

```
    If AND (96, 64) = 64 PRINT "TRUE" ELSE PRINT "FALSE"
```

Prints "TRUE".

Sample Program

```
10 REM        *** AND FUNCTION ***
20 INPUT PROMPT="ENTER AN INTEGER VALUE (-32768 TO 32767) "; X%
30 PRINT "LSB IS "; AND(X%,&00FF)
40 GOTO 20
```

-- FUNCTION --

ASC
Get ASCII Code

```
ASC(string)
    'string' is a string constant or a string variable.
```

ASC returns the ASCII code of the first character in the 'string'. The ASCII codes are listed in the Appendix.

Examples

```
   PRINT ASC("A")
   PRINT ASC("AB")
```

Both lines will print 65, the ASCII code for "A".

```
   X = ASC(B$)
```

Assigns the ASCII code for B$ to X.

Sample Program

```
  80 REM      *** SAMPLE PROGRAM DEMONSTRATING ASC ***
  90 REM
 100 REM      *** CHANGING THE OUTPUT OF A CHARACTER ON YOUR KEYBOARD ***
 110 REM
 120 PRINT "TYPE THE CHARACTER YOU WANT TO CHANGE"
 130 INPUT A$
```

```
140 PRINT "TYPE THE CHARACTER YOU WANT IT TO REPRESENT"
150 INPUT B$
160 PRINT "NOW TYPE ANY CHARACTERS ON YOUR KEYBOARD"
170 PRINT "NOTICE THAT YOUR CHARACTER HAS BEEN CHANGED"
180 C$ = INKEY$: IF C$ = "" THEN 180
190 IF C$ = A$ THEN C$ = CHR$(ASC(B$))
200 PRINT C$;
210 GOTO 180
```

-- FUNCTION --

ATN
Compute Arctangent

```
ATN(number)
  'number' is a numeric expression
```

ATN returns the angle of the 'number'. The number is the tangent. The angle will be in radians. To convert to degrees, multiply ATN(X) by 57.295779513082.

The result is always a real number.

Examples

```
   X = ATN(Y/3)
```

Assigns the value of the arctangent of Y/3 to X.

```
   PRINT ATN(1.0023) * 57.2
```

Prints 44.9905.

```
   R = N * ATN(-20 * F2/F1)
```

Assigns the indicated value to R.

NOTE: Trigonometric functions are not loaded when you load the BASIC Compiler; they are loaded upon demand. This might cause a slight delay when using these functions, since they must be loaded into the system first.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING ATN ***
 90 REM
100 PRINT "INPUT TANGENT"
110 INPUT T
120 PRINT "ANGLE IS"; ATN(T) * 57.29578
130 GOTO 100
```

-- STATEMENT --

CALL
Execute External Subroutine

```
CALL "subname"; data list
'subname' is a 1-6 character string constant
'data list' consists of any of the following
    separated by commas:
    numeric expression
    string variable
    string constant
    subscripted variable
```

A CALL statement instructs the computer to run a subprogram. In addition, it sends the list of data that you specify to the subprogram. The subprogram performs its operations on this data and sends the resulting values back to the main program.

A subprogram like an internal subroutine is called from the main program*, executed, and returns to the main program. It may be as many lines as you want and may have its own local variables, independent of the main program.

A subprogram has the added flexibility of performing the same operations on whatever data is sent to it by the main program. This is especially helpful if you are performing the same complicated computations with different variables repeatedly in different parts of your program.

CALL will not "Load" or "Old" a subprogram. All subprograms must be Loaded or Appended into memory before the main program is executed.

CALL may also be used to call an external machine language routine. To do this, you must have an EXT statement in your program defining the memory address of the routine. See EXT and the chapter on Segmenting Programs.

----

* A subprogram may also be CALLed from another subprogram.

Examples

If you have a subprogram beginning with the statement:

```
   SUB "ADD"; X, Y$
```

The following CALL statements could be used:

```
   CALL "ADD"; 5, "HEADS"
```

Executes the subprogram named "ADD". This statement also passes the data 5 and "HEADS" to the subprogram. The subprogram assigns 5 to X and "HEADS" to Y$. It then performs its routine on this data.

```
   CALL "ADD"; A, B$
```

This statement also executes the subprogram "ADD". It passes the data A and B$ to the subprogram. The subprogram assigns the value of A to X and B$ to Y$, performs its operations on X and Y$, and sends the resulting values back to the main program as A and B$.

If a subprogram begins with the statement:

```
   SUB "CHART"; M( ), N$( , )
```

Then:

```
   CALL "CHART"; C( ), D$( , )
```

Executes the subprogram "CHART" sending all the data in the one-dimensional array C and the two-dimensional array D$ to the subprogram. The subprogram performs its routine on the data and sends the resulting data back to the main program.

```
   CALL "CHART"; SALES( ), ITEMS$( , )
```

Executes the same subprogram CHART, which will perform the same routine on all the data in the SALES and ITEMS$ arrays and send the resulting data back to the main program.

NOTE: For information on how to use subprograms, see the section on Segmenting Programs. See also END, SUB, and SUBEND.

Sample Programs

```
    80 REM      *** SAMPLE PROGRAM #1 DEMONSTRATING CALL ***
    90 REM
   100 X = 2 : Y = 3 : Z = 4
   110 CALL "SUBPROG"; X
   120 CALL "SUBPROG"; Y
   130 CALL "SUBPROG"; Z
   140 PRINT X,Y,Z
   150 END
   160 SUB "SUBPROG"; A
   170 A = A * 2
   180 SUBEND
```

```
    80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING CALL ***
    90 REM
   100 PRINT "INPUT WEEKLY GROCERY EXPERNSES"
   110 INPUT F
   120 CALL "ANNUAL"; F
   130 PRINT "INPUT WEEKLY GASOLINE EXPENSES"
   140 INPUT G
   150 CALL "ANNUAL"; G
   160 PRINT "ANNUAL EXPENSES ARE ---- "
   170 PRINT F; "FOR GROCERIES", G; "FOR GASOLINE"
   180 END
   190 SUB "ANNUAL"; X
   200 X = X*52
   210 SUBEND
*RUN
INPUT WEEKLY GROCERY EXPERNSES
? 32
INPUT WEEKLY GASOLINE EXPENSES
? 20
ANNUAL EXPENSES ARE ----
 1664 FOR GROCERIES             1040 FOR GASOLINE
 80 REM      *** SAMPLE PROGRAM #3 DEMONSTRATING CALL ***
 90 REM
100 DIM U(12)
110 DIM O(12)
120 FOR I = 1 TO 12 : READ U(I) : NEXT I
130 FOR I = 1 TO 12 : READ O(I) : NEXT I
140 CALL "CHART"; "UTILITIES", U( )
150 CALL "CHART"; "OFFICE SUPPLIES", O( )
160 DATA 150,175,200,120,130,220,145,180,190,200,135,145
170 DATA 100,75,65,93,104,120,110,92,88,90,70,60
180 END
190 SUB "CHART"; A$, B( )
200 DIM C$(12)
210 PRINT CHR$(27)
220 PRINT CRT(0,30); "EXPENSES --- "; A$
230 PRINT: PRINT: PRINT
240 FOR I = 1 TO 12
250     READ C$(I): X = B(I)/3
260     PRINT C$(I); " ";
270     PRINT STRING$(X,"X")
280 NEXT I
290 PRINT CRT(20,0); "PRESS <ENTER>";
300 A$ = INPUT$(1)
310 DATA "JAN","FEB","MAR","APR","MAY","JUN","JUL","AUG","SEP"
320 DATA "OCT","NOV","DEC"
330 SUBEND
```

-- STATEMENT --

CHAIN
Load and Execute Next Program

```
CHAIN "filespec"
  'filespec' is a string constant or a string variable
representing a TRSDOS file specification
```

CHAIN loads a program stored on disk into memory and executes it. When the chained program is loaded, the resident program is deleted from memory.

Note: See also COM and the chapter on Segmenting Programs.

Examples

```
CHAIN"NEXT/BAS"
```

Loads the program NEXT/BAS and executes it.

```
CHAIN"PROG2/CMP:1"
```

Loads the program PROG2/CMP from the diskette in drive 1 and executes it.

```
CHAIN A$
```

Loads the filespec A$ and executes it.

Sample Program

```
10 REM      *** SAMPLE PROGRAM DEMONSTRATING CHAIN ***
20 REM      *** PROG2/BAS MUST FIRST BE SAVED ON DISK ***
30 REM
40 PRINT "ENDING PROGRAM 1 ---- BEGINNING PROGRAM 2"
50 CHAIN "PROG2/BAS"
```

-- FUNCTION --

CHR$
Get Character for ASCII or Control Code

```
CHR$(number)
    'number' is a numeric expression in the range
       -32768 to 32767.
```

CHR$ is the inverse of the ASC function. By specifying an ASCII code, CHR$ returns the code's corresponding one-character string. This one-character string may either be one of the keys on your keyboard or a control character. A list of ASCII codes is in the Appendix.

Note: To produce graphics characters, see CRTG

Examples

```
   PRINT CHR$(35)
```

Prints a # on the display.

```
   P$ = CHR$(T)
```

The number represented by T is converted into its ASCII character equivalent assigned to P$.

```
   PRINT CHR$(26)
```

Puts the Display into its black-on-white mode. (Use CHR$(25) to return to normal).

```
   A$ = A$ & CHR$(I)
```

The character whose ASCII code is I is added to the end of A$.

Sample Programs

```
 80 REM      *** SAMPLE PROGRAM #1 FOR CHR$ ***
 90 REM
100 PRINT CHR$(27)
110 PRINT "TYPE IN THE CODE(0-127)"
120 INPUT C
130 PRINT CHR$(C); "    JUST PRINTED THE CODE "; C
140 GOTO 110
```

```
   80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING CHR$ ***
   90 REM
  100 REM      *** 1 IS THE ASCII CODE FOR <F1> KEY; 2 IS THE CODE FOR <F2>
  110 REM
  120 PRINT "TYPE SENTENCES ON YOUR KEYBOARD"
  130 PRINT "PRESS <F1> WHEN YOU WANT A WHITE BACKGROUND"
  140 PRINT "PRESS <F2> FOR A NORMAL DISPLAY"
  150 A$ = INKEY$ : IF A$ = "" THEN 150
  160 IF A$ = CHR$(1) THEN A$ = CHR$(26)
  170 IF A$ = CHR$(2) THEN A$ = CHR$(25)
  180 PRINT A$;
  190 GOTO 150
*RUN
TYPE SENTENCES ON YOUR KEYBOARD
PRESS <F1> WHEN YOU WANT A WHITE BACKGROUND
PRESS <F2> FOR A NORMAL DISPLAY
A LONG, LONG TIME AGO, IN A GALAXY FAR, FAR AWAY, THERE LIVED A GIRL WHO GOT TIR
ED OF TYPING IN SENTENCES THAT ALWAYS START WITH A LONG, LONG TIME AGO, IN A GA
LAXY FAR, FAR AWAY.
```

-- STATEMENT --

CLOSE
Close Disk File

```
CLOSE #file-unit
  'file-unit'  is a numeric expression specifying
      which file is to be closed. If 'file-unit' is
      omitted, all open files are closed. If a
      specified file unit is not open, an error
      occurs.
```

This statement closes access to the file or files referenced by 'file-unit', assigned when the file is opened.

Examples

```
     CLOSE #1
```

Closes file-unit 1.

```
    CLOSE #START + NCRMT
```

Close file-unit (START + NCRMT).

```
    CLOSE
```

Closes all open file-units.

Sample Program

See the chapter on data files.

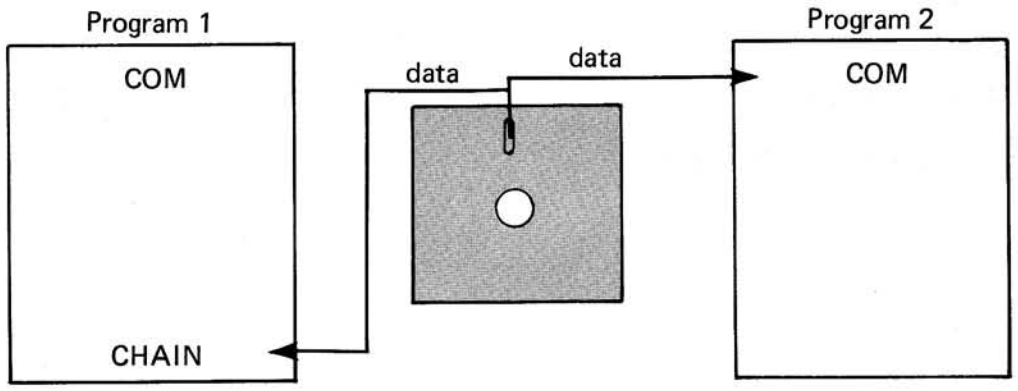
-- STATEMENT --

COM
Allocate Common Variable Area

```
COM variable list
  'variable list' is one or more variables separated
     by commas.  Each variable may be a:
     numeric variable
     string variable
     numeric array
     string array
```

You may use COM to pass one or more variables to the next program. COM allocates a common area in the program for variables so that they may be passed to the next program.

Note: See also CHAIN and the chapter on Segmenting Programs.



Examples

```
   COM C, D$
```

Allocates a common area for storing the variables

C and D$ so they may be accessed by the next program.

```
   COM B$(50)
```

Allocates a common area for storing array B$ with 51 elements (0-50) so that the array may be accessed by the next program.

```
   COM A(10,10)
```

Allocates a common storage area for the two dimensional array A.

Sample Program

```
 30 REM     *** SAMPLE PROGRAM DEMONSTRATING COM ***
 40 REM
 50 REM     *** PROG2/BAS MUST FIRST BE SAVED ON DISK ***
 60 REM
 70 REM     *** PROG2/BAS WILL RETAIN WHATEVER VALUES   ***
 80 REM     *** THIS PROGRAM SETS FOR A$ AND B      ***
 90 REM
100 COM A$, B
110 PRINT "INPUT A NAME AND A NUMBER"
120 INPUT A$, B
130 CHAIN "PROG2/BAS"
```

-- FUNCTION --

COS
Compute Cosine

```
COS(number)
   'number' is a numeric expression.
```

COS returns the cosine of the 'number'. The 'number' should be an angle, which must be given in radians. When the 'number' is in degrees, use COS('number' * .01745329251993).

The result is always a real number.

Examples

```
Y = COS(X)
```

Assigns the value of COS(X) to Y.

```
Y = COS(X * .01745329251994)
```

If X is an angle in degrees, the above line will give its cosine.

```
PRINT COS(5.8) - COS(85 * .42)
```

Prints the difference of the two cosines.

```
G2 = G1 * ((COS(A)) * 15)
```

Computes the indicated cosine and stores it in G2.

NOTE: Trigonometric functions are not loaded when you load the BASIC Compiler; they are loaded upon demand. This might cause a slight delay when using these functions, since they must be loaded into the system first.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING COS ***
 90 REM
100 PRINT "INPUT ANGLE IN RADIANS"
110 INPUT A
120 PRINT "COSINE IS"; COS(A)
130 GOTO 100
```

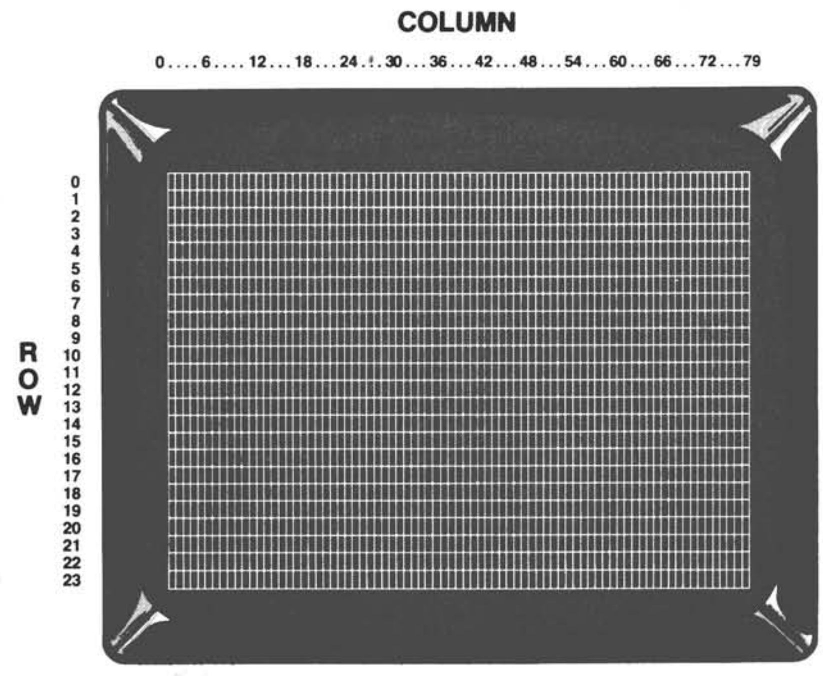
-- FUNCTION --

CRT
Position Cursor

```
CRT(row, column)
    'row' is a number between 0 and 23.  If outside
       that range BASIC performs a MOD 24.
    'column' is a number between 0 and 79.  If outside
       that range, BASIC performs a MOD 80.
```

CRT, used in a PRINT statement, positions the cursor at the 'row' and 'column' specified on the video display. It may only be used in a PRINT statement.

Note: The Model II video display consists of 24 rows (0 to 23) and 80 columns (0 to 79):



'row' and 'column' refer to a row and column on the video display.

Examples:

```
   PRINT CRT(0,79);"&"
```

Positions the cursor at the top right hand corner and prints "&".

```
   PRINT CRT(23, 0);"THIS IS LOCATION 23, 0"
```

Positions the cursor at the bottom left-hand corner of the display and prints the message beginning at that position.

```
   PRINT CRT(25, 0);"###"
```

Positions the cursor at the beginning of row 1 in position 1,0 and prints ###. (Since 25 is outside the range 0 -23, BASIC performs a MOD 24 and reduces the 25 to a 1).

Sample Program

```
10 PRINT CHR$(27)
20 PRINT "WHAT IS YOUR LAST NAME"
25 PRINT CRT(2,0);
30 INPUT A$
40 PRINT CRT(8,0); "YOUR FIRST NAME"
45 PRINT CRT(10,0);
50 INPUT B$
60 PRINT CRT(14,10); "THANK YOU, "; B$; " "; A$; "!"
```

```
WHAT IS YOUR LAST NAME
? WILLIAMS



YOUR FIRST NAME

? SANDY


          THANK YOU, SANDY WILLIAMS!
```

-- FUNCTION --

CRTG
Print in Graphics Mode

```
CRTG (row, column, string)
    'row' is a whole number in the range of [0,32767].
       If larger than 23, BASIC reduces it by MOD 24.
    'column' is a whole number in the range of
       [0,32767].  If larger than 79, BASIC reduces it
       by MOD 80.
    'string' is a string constant or a string variable.
```

CRTG used in a PRINT statement, prints 'string' in the graphics mode. The 'string' is printed as follows:

1. The first character of the string is printed at the 'row', and 'column' position specified.

2. The cursor is then advanced to the next column position on the same row. If the next position is 80, the cursor wraps the display to column 0 of the next row. If the next row is 24, the cursor wraps the display to row 0.

3. The next character in the string, if there is one, is then printed at the cursor position. Steps 2 and 3 are then repeated.

Note: See CRT for an illustration of 'row' and 'column'positions on the video display.

The 'string' may contain up to 255 characters which may be printed in graphics mode. The characters are listed in the Appendix. The first 32 are special graphics characters. The rest are alphanumeric or control characters.

As shown in the listing, all of the alphanumeric characters may be referenced either by the keyboard character itself, or by the character's ASCII code. For example:

```
A$ = "M"
A$ = CHR$(77)
```

both assign the character M to A$.

Special graphics characters may be referenced by a control character on the keyboard, or by the character's ASCII code:

```
   A$ = "<CTRL><T>"
   A$ = CHR$(20)
```

both assign the special graphics character which looks like a bar to A$.

Note: In this example, the <CTRL> and <T> keys should be pressed simultaneously.

The easiest way to print graphics images on the display is to build a string of graphics characters. For example:

```
   10  A$ = CHR$(12)
   20  B$ = CHR$(30)
   30  C$ = B$&A$&A$&A$&A$&B$
   40  PRINT CHR$(27)
   50  PRINT CRTG(0,0,C$)
```

Prints an image which looks like a railroad track at the top left hand corner of the screen.

The sample programs for CRTG illustrate different ways of printing in the graphics mode.

Note: See also CRT, PRINT, and CHR$

Examples

```
   PRINT CRTG(23,0,C$)
```

Prints the contents of string C$ at the bottom left hand corner of the display.

```
   PRINT CRTG(11,40,"<CTRL Z>")
```

Prints a tiny square in the center of the display.

```
   PRINT CRTG(11,40,"X")
```

Prints an X in the center of the display.

Sample Programs

```
  5 DIM CHAR$120
 10 FOR I% = 0 TO 127
 20 CHAR$ = CHAR$ & CHR$(I%)
 40 NEXT I%
 50 PRINT CRTG(CRTX, CRTY, CHR$)
```

```
 10 REM         CRTG DEMO: USE OF GRAPHICS FOR GRAPHS
 20 INTEGER
 30 DIM TOP$40,BOT$41,MDL$40,CD0$1,CD1$1,CD2$1
 40 CD0$ = CHR$(24): CD1$ = CHR$(27): CD2$ = CHR$(15)
 50 TOP$ = STRING$(40, CD0$)
 60 BOT$ = CHR$(250) & STRING$(40,CD1$) & CHR$(249)
 70 MDL$ = CHR$(11) & STRING$(38,CD2$) & CHR$(09)
 80 PRINT CHR$(27)     :REM  CLEAR SCREEN
 90 PRINT CRTG(2,20,TOP$);
100 FOR LIN = 3 TO 20
110      PRINT CRTG(LIN,20,MDL$);
120 NEXT LIN
130 PRINT CRTG(21,20,BOT$);
140 PRINT CRT(0,0);"TYPE TO CHANGE GRAPH; USE ARROW KEYS FOR CURSOR CONTROL"
150 PRINT CRT(3,20); CHR$(26);    : REM POSITION CURSOR AND REVERSE MODE
160 ON BREAK GOTO 260
170 CD0$ = INKEY$
180 IF CD0$ = "" THEN 170
190 IF (CHR$(27) < CD0$) AND (CHR$(32) > CD0$) THEN 220
200 PRINT CD0$;
210 GOTO 170
220 CD0$ = CHR$(ASC(CD0$)+224)
230 PRINT CRTG(CRTX,CRTY,CD0$);
240 GOTO 170
260 PRINT CHR$(25);     : REM  NORMAL MODE
```

```
 4 DIM A$1
 8 PRINT CHR$(27)
10 PRINT "ENTER GRAPHICS CODE (1-32)"
20 INPUT CODE%
30 A$ = CHR$(CODE%)
40 PRINT CRTG(12,40,A$)
50 GOTO 10
```

-- FUNCTION --

CRTI$
Read Video Display

```
CRTI$(row, colomn, length)
    'row' is a row on the video display from 0 to 23
    'column' is a column on the video display from
       0 to 79
    'length' is the number of characters you want
       read into the string.
```

CRTI$ reads the characters on the video display in the area of the display that you specify. It returns a string of characters beginning on 'row' and 'column' with the length that you specify.

Note: See CRT for an illustration of row and column positions.

Examples

If, immediately before executing the statements below, this is printed on your video display beginning at position row 1, column 0:

```
(c) 1979 by Ryan-McFarland Corp.  All rights reserved.
```

Then:

```
PRINT CRTI$(1,0,10)
```

Prints "(c) 1979 b"

```
A$ = CRTI$(1,0,54)
```

Stores "(c) 1979 by Ryan-McFarland Corp. All rights reserved." in A$.

```
PRINT CRTI$(1,12,42)
```

Prints "Ryan-McFarland Corp. All rights reserved."

Sample Programs

```
 80 REM      *** SAMPLE PROGRAM #1 DEMONSTRATING CRTI$ ***
 90 REM
 95 REM      *** PROGRAM TO PRINT VIDEO DISPLAY TO THE LINE PRINTER ***
 96 REM
100 DIM A$80(23)
110 FOR Z = 0 TO 23
120     A$(Z) = CRTI$(Z,0,80)
130     LPRINT A$(Z)
140 NEXT Z
```

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING CRTI$ ***
 90 REM
100 PRINT CHR$(27)
110 PRINT "TYPE IN ONE LINE OF TEXT"
120 PRINT CRT(3,0);
130 A$ = INPUT$(80)
140 PRINT:PRINT:PRINT
150 PRINT "THIS IS THE LINE YOU TYPED: "
160 PRINT : PRINT CRTI$(3,0,80)
100 PRINT CHR$(27)
110 PRINT "WHAT IS YOUR FIRST NAME"
120 PRINT CRT(12,25)
130 A$ = INPUT$(10)
140 PRINT CRT(2,0); "YOUR LAST NAME"
150 PRINT CRT(12, LEN(A$) + 26)
160 B$ = INPUT$(10)
170 PRINT CRT(23,60); "THANK YOU"
180 GOTO 180
```

```
 80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING CRTI$ ***
 90 REM
100 INTEGER A-Z
110 DIM V$80(24)
120 PRINT CHR$(27);
130 PRINT "TYPE IN AS MUCH AS YOU WISH--PRESS <F1> TO STORE DISPLAY"
140 A$ = INKEY$: IF A$ < " " THEN 140
150 PRINT CHR$(27); A$;
160 A$ = INKEY$: IF A$ < " " THEN  190
170 PRINT A$;
180 GOTO 160
190 REM CHECK FOR VALID CONTROL KEY
200 IF A$ = CHR$(8) OR A$ = CHR$(13) THEN 170
210 IF A$ = CHR$(1) THEN 230
220 GOTO 160
230 REM *** READ VIDEO ***
240 ROW = CRTX: COL = CRTY
250 FOR LN = 0 TO ROW - 1
260     V$(LN) = CRTI$(LN, 0, 80)
270 NEXT LN
280 V$(ROW) = CRTI$(ROW, 0, COL)
290 PRINT CHR$(27); "TEXT STORED--PRESS ANY KEY TO SEE IT"
300 A$ = INPUT$(1)
310 FOR LN = 0 TO ROW
320     PRINT V$(LN);
330 NEXT LN
```

-- FUNCTION --

CRTR
Move Cursor

```
CRTR(row,column)
    'row' is a number in the range of [0,32767]
    'column' is a number in the range of [0,32767]
```

CRTR may only be used in a PRINT statement. PRINT CRTR makes the cursor move in relation to its present position on the video screen. If this causes the cursor to "move off the display", the cursor will wrap around.

CRTR works by performing this calculation:

```
    the number of 'rows' and 'columns' you specify
 +  the cursor's present row and column position
 -----------------------------------------------
    the cursor's new row and column position
```

If the sum of the rows is greater than 24, BASIC will perform a MOD 24. If the sum of the columns is greater than 79, BASIC will perform a MOD 80.

For example, if the cursor is presently at row 20, column 70, and you execute a CRTR(10,20) statement, BASIC will compute the sum of the two rows and the two columns:

```
                                Row          Column
CRTR specification:              10              20
Present cursor position:        +20            + 70
                                 --              --
Totals:                          30              90
```

The results are both outside the range of the video screen. BASIC will then perform a MOD 24 on the row total (30 / 24 = 1 remainder 6) and a MOD 80 on the column total (90 / 80 = 1 remainder 10). The result of this is row 6, column 10.

Note: See CRT for an illustration of row and column positions.

Examples

If the cursor is currently at row 20, column 50 ----

```
   PRINT CRTR(2, 10)
```

causes the cursor to more to row 22, column 60.

```
   PRINT CRTR(2, 10);"X"
```

causes the cursor to move to row 22, column 60. It prints the X at the next column position -- row 22, column 61.

```
   PRINT CRTR(4,40);"****"
```

causes the cursor to wrap around to row 0, column 10. The **** is printed at beginning at the next column position -- row 0, column 11.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING CRTR ***
 90 REM
100 PRINT CHR$(27)
110 PRINT CRT(0,0);"X";
120 PRINT CRTR(1,0);"X";
130 FOR I = 1 TO 50 :              REM  *** THESE TWO LINES SET A PAUSE ***
140 NEXT I :                       REM  *** AFTER EACH X IS PRINTED ***
150 GOTO 120
```

-- FUNCTION --

CRTX
CRTY
Find Cursor Position

```
CRTX
CRTY
```

CRTX returns the row and CRTY returns the column of the current cursor position.

Note: See CRT for an illustration of row and column positions.

Examples

If the cursor is currently on row 10, column 15 of the video display:

```
   R = CRTX
```

Stores 10 in R

```
   C = CRTY
```

Stores 15 in C

```
   PRINT "CURSOR IS IN ROW "; CRTX; " COLUMN "; CRTY
```

Prints CURSOR IS IN ROW 10 COLUMN 15.

Sample Program

```
 80 REM   *** SAMPLE PROGRAM DEMONSTRATING CRTX, CRTY ***
 90 REM
100 PRINT CHR$(27)
110 PRINT "TYPE AN <X> ANYWHERE ON THE SCREEN --"
120 PRINT "YOU MAY USE <SPACE BAR> AND <ENTER> TO POSITION CURSOR"
```

```
 130 A$ = INKEY$
 140 PRINT A$;
 150 IF A$ <> "X" THEN 130
 160 PRINT
 170 PRINT "YOUR <X> IS ON ROW"; ;CRTX; " AND COLUMN"; CRTY
```

```
TYPE AN <X> ANYWHERE ON THE SCREEN ---
YOU MAY USE <SPACE BAR> AND <ENTER> TO POSITION CURSOR



                                                X
YOUR <X> IS ON ROW 7  AND COLUMN 0
```

-- FUNCTION --

CVD
Convert to Real Value

```
CVD(number)
  'number' is an integer in the range of [-32768,32767]
```

CVD converts the 'number' to a real number.

Examples

```
PRINT CVD(30000) + CVD(10000)
```

Converts 30000 and 10000 to real numbers, performs real number addition, and gives the correct answer. (See explanation on numeric operations in the chapter on BASIC Concepts).

Sample Program

```
  80 REM     *** SAMPLE PROGRAM DEMONSTRATING CVD ***
  90 REM
 100 PRINT "SINCE 30000 IS AN INTEGER"
 110 PRINT "BUT 60000 IS OUTSIDE THE INTEGER RANGE"
 120 PRINT "THE PROBLEM 30000 + 30000 CAUSES THIS TO HAPPEN ..."
 130 PRINT "30000 + 30000 = "; 30000 + 30000
 140 PRINT
 150 PRINT
 160 PRINT "USING CVD TO CONVERT BOTH OPERANDS TO REAL NUMBERS"
 170 PRINT "THE PROBLEM IS SOLVED CORRECTLY ..."
 180 PRINT "30000 + 30000 = "; CVD(30000) + CVD(30000)
*RUN
```

```
SINCE 30000 IS AN INTEGER
BUT 60000 IS OUTSIDE THE INTEGER RANGE
THE PROBLEM 30000 + 30000 CAUSES THIS TO HAPPEN ...

NUMERIC OVERFLOW ERROR LINE 130
30000 + 30000 =  32767


USING CVD TO CONVERT BOTH OPERANDS TO REAL NUMBERS
THE PROBLEM IS SOLVED CORRECTLY ...
30000 + 30000 =  60000
```

-- FUNCTION --

CVI
Convert to Integer Representation

```
CVI(number)
  'number' is a numeric expression in the range of
     -32768 to 32767.
```

CVI returns the largest integer not greater than the 'number'. For example, CVI(1.5) returns 1; CVI(-1.5) returns -2. The result is always a two-byte integer.

Since integers are stored in two bytes and real numbers are stored in eight bytes, converting a number to its integer representation changes its storage format. BASIC will execute numeric operations, such as addition, subtraction, multiplication, and division, much more quickly with integers than with real numbers.

Examples

```
   PRINT CVI(15.0075)
```

Prints 15.

```
   PRINT CVI(-15.0075)
```

Prints -16.

```
    PRINT CVI(6.1 + 2.2)
```

Prints 8.

```
    A = CVI(X)
```

Assigns the integer representation of X to A.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING CVI ***
 90 REM
100 PRINT "ENTER A NUMBER WITH A FRACTIONAL VALUE (LIKE DDDD.DDDD)"
110 INPUT N
120 PRINT "INTEGER PORTION IS"; CVI(N)
130 GOTO 100
```

-- STATEMENT --

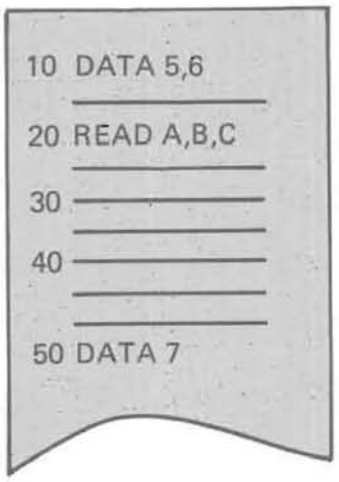
DATA
Store Program-Data

```
DATA item-list
    'item list' is a list of string and/or numeric
        constants, separated by commas.  String
        constants must be in quotes.
```

The DATA statement lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement.

DATA statements may appear anwhere it is convenient in the program. Generally, they are placed together, but this is not required. It is important that the types of data match up with the corresponding variable types in the READ statement.

The data in DATA statements may only be constants. No variables or expresssions are allowed.



Examples

```
   DATA "NEW YORK","CHICAGO","LOS ANGELES","PHILADELPHIA"
```

This line contains four string data items.

```
   DATA 3.72,3.14159,47.29578,378,535
```

This line contains five numeric data items.

```
   DATA "SMITH, T.H.",38,"THORN,J.R.",41
```

This line contains two string and two numeric data items.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING DATA ***
 90 REM
100 DIM SALES(6)
110 FOR X = 1 TO 6
120    READ DEPT$
130    PRINT "INPUT AMT. SOLD IN THE "; DEPT$; " DEPT. :";
140    INPUT SALES(X)
150 NEXT X
160 DATA "PRODUCE","MEAT","BAKERY","CANNED GOODS","DAIRY","FROZEN FOODS"
```

-- FUNCTION --

DATE$
Get Today's Date

```
DATE$
```

This function lets you display today's date and use it in the program.

The operator sets the date initially when TRSDOS is started up. When you request the date, BASIC will display it in the fashion:

```
SATAPR281979118 45
```

which means Saturday, April 28, 1979, 118th day of the year, 4th month of the year, 5th day of the week (Monday is the 0th day of the week.

Example

```
PRINT DATE$
```

which returns:

```
MONAPR171980108 40
```

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING DATE$ ***
 90 REM
100 PRINT DATE$
110 PRINT "INVENTORY CHECK: "
120 IF DATE$ = "THUJAN311980 31" THEN PRINT "Today is the last
                  day of January 1980.  Time to perform monthly in
                  ventory": STOP
```

```
130 D$ = DATE$ : A$ = SEG$(D$, 7, 2)
140 B = VAL(A$)
150 PRINT 31 - B; " days until inventory time."
```

-- STATEMENT --

DEF
Define Function

```
DEF function name(dummy variable, ...) = formula
   'function name' is any valid variable name.
   'dummy variable' is any valid variable name which
      the formula will perform operations on.
   'formula' is a numeric or string expression usually
      involving the 'dummy variable(s)' on the left side
      of the equals sign.
```

The DEF statement lets you create your own function. Once you have defined the operations your function will do, all you have to do is call the new function by name and the operations will be automatically performed. To call it by name, after it has been defined with the DEF statement, simply reference the 'function name' in an expression. You can use it exactly as you might use one of the built-in functions, like SIN, ABS and STRING$.

The type of variable used for function name determines the type of value the function will return. For example, if 'function name' is an integer variable, then that function will return an integer even if the data used in the function are real numbers.

You may pass any data with the same type of value to the 'dummy variable'. Furthermore, you may use the same variable name as the 'dummy variable' in your program without the 'dummy variable' interfering with your program variables.

Examples

```
   DEF R(A) = INT(RND(0) * (A) + 1)
```

This statement defines a function which returns a random whole number between 1 and A. The value for A is passed in a statement using R such as this:

```
   Y = R(X)
```

If X equals 10, a random whole number between 1 and 10 will be assigned to Y.

```
   DEF SL$(X) = STRING$(X, "-")
```

Defines the function names SL$ which returns a string of hyphens X characters long. The value for X is passed in a statement using SL$ such as:

```
   PRINT SL$(30)
```

Which prints a string of 30 hyphens.

```
   DEF DIV(X,Y) = SQR(X)/SQR(Y)
```

Defines a function named DIV which divides the square root of X by the square root of Y. It can be used like this:

```
   PRINT DIV(100, 25)
```

Which prints 2.

Sample Programs

```
 80 REM      *** SAMPLE PROGRAM #1 DEMONSTRATING DEF    ***
 90 REM
100 DEF DOUBLE(N) = N * 2
110 PRINT "INPUT A NUMBER"
120 INPUT N
130 PRINT DOUBLE(N)
140 GOTO 110
```

```
 80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING DEF  ***
 90 REM
100 DEF SOUND(T) = (1087 + SQR(273 + T))/16.52
110 PRINT "INPUT AIR TEMPERATURE IN DEGREES CELSIUS"
120 INPUT T
130 PRINT "THE SPEED OF SOUND IN AIR OF "; T; "DEGREES CELSIUS IS"
140 PRINT SOUND(T); "FEET PER SECOND"
```

-- STATEMENT --

DELETE
Delete Record From Disk File

```
DELETE #file-unit, KEY = record
 'file-unit' specifies the file in terms of the
     'file-unit' assigned when the file was
         opened.
 'KEY = record' specifies which record is to
     be deleted; for ISAM records, 'record'
     is a string expression; for direct-access
     records, it is a numeric expression.
```

This statement deletes a record from a disk file. After a record has been deleted, it is unreadable.

Examples

```
   DELETE #1, KEY=2
```

Deletes the 2nd record in file-unit #1.

```
   DELETE #A%, KEY=NAMES$
```

Deletes in file-unit A% the ISAM record with a key matching the value of NAME$.

```
    DELETE #START% + INC%, KEY=RECORD%
```

Deletes in file-unit START% + INC% the record numbered as RECORD%.

Sample Program

See the chapter on data files.

-- FUNCTION --

DIG
Compute Number of Numeric Characters

```
DIG(string)
   'string' is a string constant or a string variable.
```

DIG computes the number of numeric characters in the 'string'. It will quit searching for numeric characters as soon as it hits a non-numeric character. For example, in DIG("16A5"), DIG will quit counting numeric characters when it reaches the A, since A is non-numeric, and will return the current total, 2.

DIG treats blanks, signs, decimals, and exponents as numeric characters.

Examples

```
   PRINT DIG("1.2E5")
```

Prints 5

```
   PRINT DIG("33 44")
```

Prints 5. (The blank is considered part of the numeric field).

```
   A = DIG("-32")
```

Prints 3.

```
   X = DIG(B$)
```

Assigns the number of numeric characters in B$ to X.

```
   PRINT DIG("B5")
```

Prints 0. (DIG quits searching for numeric characters after it reads the non-numeric character, B).

```
   PRINT DIG("5B324")
```

Prints 1.

Sample Program

```
    100 REM     *** DEMO OF DIG FUNCTION TO EDIT A STREAM OF DATA
    200 REM
    205 REM   T$       CONTAINS THE INPUT STREAM
    210 REM   MAXPSN% CONTAINS THE LENGTH OF THE INPUT STREAM
    220 REM   PSN%     POINTS TO THE CURRENT START-EDIT POSITION
    230 REM   CRNT$    CONTAINS THE CURRENT STRING TO BE EDITED
    240 REM   VLULEN   IS THE LENGTH OF THE FIRST NUMERIC FIELD
    250 REM              A ZERO LENGTH INDICATES A NON-NUMERIC FIELD
    260 REM   VLU      VALUE OF THE FIRST NUMERIC FIELD
    270 REM
    300 DIM T$80, CRNT$80
    400 PRINT "ENTER A STREAM OF NUMBERS, SEPARATED BY COMMAS"
    500 LPRINT "ENTER A STREAM OF NUMBERS, SEPARATED BY COMMAS"
    600 LINE INPUT T$
    700 LPRINT T$
    800 MAXPSN% = LEN(T$)
    900 PSN% = 1
   1000 CRNT$ = SEG$(T$, PSN%)
   1100 VLULEN% = DIG(CRNT$)
   1200 IF VLULEN% = 0 THEN 1600
   1300 VLU = VAL(CRNT$)
   1400 PRINT "FOUND THIS NUMBER: "; VLU
   1500 LPRINT "FOUND THIS NUMBER: "; VLU
   1600 PSN% = PSN% + VLULEN$ + 1
   1700 IF PSN% > MAXPSN% THEN PRINT:LPRINT: GOTO 400
   1800 GOTO 1000
```

```
ENTER A STREAM OF NUMBERS, SEPARATED BY COMMAS
2, 23, 34, 45, 5, 67, 678, 56, 6789, 456
FOUND THIS NUMBER:  2
FOUND THIS NUMBER:  23
FOUND THIS NUMBER:  34
FOUND THIS NUMBER:  45
FOUND THIS NUMBER:  5
FOUND THIS NUMBER:  67
FOUND THIS NUMBER:  678
FOUND THIS NUMBER:  56
FOUND THIS NUMBER:  6789
FOUND THIS NUMBER:  456

ENTER A STREAM OF NUMBERS, SEPARATED BY COMMAS
```

-- STATEMENT --

DIM

Define String Variables and Arrays

```
DIM variable list
   'variable list' can consist of the following
       separated by commas:
       string variable length
          'string variable' is any valid string
             variable name
          'length' is an integer constant specifying
             the maximum number of characters
             in string variable
       array string length(subscript1, subscript2)
           'string length' is the length of each
              element in a string array.  If omitted,
              each element will be stored as 255
              characters.  'string length' is omitted
              in numeric arrays.
           'array' is any valid variable name
           'subscript1' and 'subscript2' are integer
              constants specifying the maximum
              number of subscripts in that dimension
              of the array.  If subscript2 is
              omitted, it is a single dimensioned
              array.

NOTE:  the lowest element in a dimension is always 0.
```

This statement defines the length of string variables and arrays.

Defining String Variables

In Compiler BASIC, each string variable is stored according to the length specified in the STRING statement. If you do not have a STRING statement in the program, each string variable is stored as if it contains 255 characters.

To override this, you may use DIM to specify the length of a particular string variable name. For example:

```
   DIM NAME$10
```

alots 10 characters for NAME$.

Defining Arrays

An array is a way of storing an entire list of data under one variable name. Each data element is identified by one or two subscripts. If each data element is identified by only one subscript, it is called a single dimensioned array; if it is identified by two subscripts, it is a two-dimensioned array. No more than two dimensions are allowed in Compiler BASIC.

All arrays must be defined with a DIM statement before they can be used in the program. For example:

```
   DIM A(2)
```

Alots room in memory for an array named A which can contain up to 3 numeric data elements (0,1,and 2). For example, each of these subscripted variables could be assigned:

```
   A(0) = 3.5
   A(1) = 40000
   A(2) = 5.15
```

A double dimensioned array is defined in this manner:

```
   X(1,1)
```

This alots room for a double dimensioned array named X which can contain up to 2 numeric data elements in the first dimension and 2 numeric data elements in the second dimension. This real number array might be programmed to contain:

```
   X(0,0) = 25.1            X(0,1) = 13.7
   X(1,0) = 22.2            X(1,1) = 32.6
```

Arrays may also be integer or string with the proper type declaration tag. A string array will alot 255 characters for each data element unless the string length is defined. For example:

```
   A$(10)
```

Alots room for an array named A$ with up to 11 string data elements. Memory is set aside for each of the 11 data elements

to contain 255 characters for a total of 255x11=
2805 characters.

```
   A$5(10)
```

This also alots room for an array named A$ with up to 11 string data elements. However, in this array, each element may contain only 5 characters for a total of 5x11=55 characters.

Examples

```
   DIM A(100), B$5, C%(9,9)
```

The numeric array A is defined with 101 elements, and C% is defined containing 100 (10 * 10) elements. The string B$ can contain no more than 5 characters.

```
   DIM DATA$3, DAVIS$6, DVI$1
```

The strings DATA$, DAVIS$, and DVI$ are defined containing 3, 6, and 1 characters respectively.

```
   DIM M$1(200), C$2(100)
```

The array M$ is defined to contain 201 one-character string data elements. Array C$ may contain 101 two-character string data elements.

Sample Programs

```
  80 REM     *** SAMPLE PROGRAM #1 DEMONSTRATING DIM ***
  90 REM
 100 DIM A%(10,10)
 110 PRINT "SALES DATA WILL BE STORED IN ARRAY A% AS FOLLOWS"
 120 PRINT CHR$(27): PRINT " ", "MONTH 1", "MONTH 2", "MONTH 3"
 130 FOR X = 1 TO 4
 140 PRINT : PRINT "ITEM "; X,
 150     FOR Y = 1 TO 3
 160     READ A%(X,Y)
 170     PRINT A%(X,Y),
 180     NEXT Y
 190 NEXT X
 200 PRINT: PRINT "INPUT ITEM # AND MONTH #"
 210 INPUT X,Y
```

```
  220 PRINT "SALES DATA FOR ITEM "; X; " AND MONTH "; Y; " IS : "; A%(X,Y)
  230 GOTO 200
  240 DATA 34,63,55,66,33,22,11,99,88,77,66,55

                 MONTH 1          MONTH 2          MONTH 3

ITEM  1           34               63               55
ITEM  2           66               33               22
ITEM  3           11               99               88
ITEM  4           77               66               55
INPUT ITEM # AND MONTH #
? 2,2
SALES DATA FOR ITEM  2  AND MONTH  2  IS :  33

INPUT ITEM # AND MONTH #
?
```

```
   80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING DIM ***
   90 REM
  100 REM      *** DIMENSION ARRAY I$ FOR FIVE 7-CHARACTER ITEMS ***
  110 DIM I$7(5)
  120 REM      *** READ IN LIST OF ITEMS ***
  130 FOR X = 1 TO 5
  140     READ I$(X)
  150 NEXT X
  160 REM      *** SEE HOW EACH ITEM IS INDEXED ***
  170 PRINT "INPUT AN ITEM NUMBER"
  180 INPUT X
  190 PRINT I$(X)
  200 GOTO 170
  210 DATA "APPLES","ORANGES","GRAPES","PEACHES","PLUMS"
*RUN
INPUT AN ITEM NUMBER
? 3
GRAPES
INPUT AN ITEM NUMBER
? 4
PEACHES
INPUT AN ITEM NUMBER
? 2
ORANGES
INPUT AN ITEM NUMBER
?
```

```
 10 REM      *** SAMPLE PROGRAM #3 DEMONSTRATING DIM ***
 20 REM
 30 PRINT CHR$(27)
 40 DIM L$(10,3)
 50 M = 0
 60 PRINT "MEMBERSHIP ARRAY IS DIMENSIONED FOR UP TO 10 MEMBERS"
 70 M = M + 1
 80     PRINT "INPUT NAME, ADDRESS, AND PHONE # OF MEMBER "; M
 90 FOR X = 1 TO 3
100 INPUT L$(M,X)
110 NEXT X
120 IF M = 10 THEN 160
130 PRINT "IS THERE ANOTHER MEMBER (Y/N)"
140 INPUT A$
150 IF A$ = "Y" THEN 70
160 PRINT: PRINT "THE LIST IS STORED AS FOLLOWS : "
170 PRINT "NAME", "ADDRESS", "PHONE"
180 PRINT STRING$(80, "-");
190 FOR I = 1 TO M
200    FOR J = 1 TO 3
210        PRINT L$(I,J),
220    NEXT J
230    PRINT
240 NEXT I
```

-- STATEMENT --

END
Terminate Program Compilation

```
END
```

END terminates compilation of your main program. This means, when you are RUNning or COMPILEing a program, the Compiler will quit compiling and assume the program has ended as soon as it encounters an END statement. Since this is different from the way END works in the BASIC Interpreter, it is important that you remember not to use END in the middle of a program if you want to use the lines following the END statement. Use STOP for that purpose.

Some versions of BASIC require END as the last statement in a program. In Compiler BASIC this is optional. However, when using a subprogram, you must put an END statement as the last statement in your main program. Otherwise, BASIC will not be able to separate your main program from the subprogram.

Note: See also SUB, SUBEND, CALL, and the chapter on Segmenting Programs.

Example

```
   END
```

This statement "turns off" the compiling of your program. BASIC then assumes there are no more main program lines following this statement.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING END ***
 90 REM
100 PRINT "EXECUTING MAIN PROGRAM"
110 CALL  "SUBPROG"; "NOW EXECUTING THE SUBPROGRAM"
120 PRINT "BACK TO THE MAIN PROGRAM"
130 END
140 SUB "SUBPROG"; A$
150 PRINT A$
160 SUBEND
```

-- FUNCTION --

EOF
Notify if End of File

```
EOF(#file-unit)
   'file-unit' is a numeric expression specifying
       a file opened for sequential access.
```

This function tells whether the end-of-file (EOF) has been reached during sequential input. If the EOF has been reached, it returns a value of -1 (TRUE). Otherwise, it returns a value of 0 (FALSE).

Examples

```
    IF EOF(#1) = -1 THEN CLOSE #1
```
If the end of file has been reached in file-unit 1, the file is closed.

```
    STATUS% = EOF(#A%)
```
File-unit A%'s EOF status (-1/TRUE or 0/FALSE) is stored in STATUS%.

Sample Program

See Chapter 4.

-- FUNCTION --

ERR
Get Error Code

```
ERR
```

ERR returns the code of the error that happened in the program. It is normally used inside an error-handling routine accessed by ON ERROR GOTO. The section on error codes in the Appendix gives the error code for each error.

Examples

```
IF ERR = 7 THEN 1000 ELSE 2000
```

If the error is an Out of Data error (code 7) the program branches to line 1000; if it is any other error, control will instead go to line 2000.

Sample Program

```
   80 REM     *** SAMPLE PROGRAM DEMONSTRATING ERR ***
   90 REM
  100 ON ERROR GOTO 150
  110 DATA 1, 2
  120 READ A, B, C
  130 PRINT " A = "; A; " B = "; B; " C = "; C
  140 STOP
  150 IF ERR <> 7 THEN ERROR ERR
  160 PRINT "YOU DON'T HAVE ENOUGH DATA FOR ALL THE VARIABLES"
  170 GOTO 130
*RUN
YOU DON'T HAVE ENOUGH DATA FOR ALL THE VARIABLES
 A =  1  B =  2  C =  3.910< E-74
```

-- STATEMENT --

ERROR
Simulate Error

```
   ERROR code
      'code' is a numeric expression defining the
          error code
```

An ERROR statement in your program causes BASIC to act exactly as if the specified error had occurred. You can specify an error with its error code. The Appendix has a listing of error codes and their meanings.

ERROR is primarily used in ON ERROR GOTO routines: either for simulating the error that occurred or for testing the routine.

Examples

```
   ERROR 7
```

When your program reaches this line, an Out of Data error (code 7) will "occur", and the Computer will print a message to this effect.

```
   IF ERR <> 5 THEN ERROR ERR
```

This line could be in the error handling routine initiated by ON ERROR GOTO. It tells the Computer that if the error which caused it to come to this routine was not an Input Syntax error (code 5), then print the appropriate error message.

Sample Program

```
    100 INPUT N
    110 ERROR N
```

-- FUNCTION --

EXP
Compute Natural Exponential

```
EXP(number)
  'number' is a numeric expression.
```

EXP returns the natural exponential of the 'number', that is, e to the power of 'number'. This is the inverse of the LOG function; therefore, X = EXP(LOG(X)). The result is always a real number.

Examples

```
   H = EXP(A)
```

Assigns the value of EXP(A) to H.

```
   PRINT EXP(-2)
```

Prints the value .135335.

```
   E = (G1 + G2 - .07) * EXP(.055 * (G1 + G2))
```

Performs the required calculation and stores it in E.

Sample Program

```
10 PRINT "INPUT A NUMBER"
20 INPUT N
30 PRINT "E RAISED TO THE N POWER IS"; EXP(N)
40 GOTO 10
```

-- FUNCTION --

EXP10
Compute Base 10 Exponential

```
EXP10(number)
  'number' is a numeric expression
```

EXP10 raises 10 to the power of 'number'. As the inverse of LOG10, X=EXP10(LOG10(X)). The result is always a real number.

Examples

```
   X = EXP10(Y)
```

Raises 10 to the Y power and assigns that value to X.

```
   PRINT EXP10(3)
```

Prints 1000.

```
   X = (A + B) + EXP10(A)
```

Performs the calculation and records the result in X.

Sample Program

```
 5 INTEGER R
10 PRINT "TABLE OF RANDOM NUMBERS ... "
20 PRINT "ENTER MAXIMUM NUMBER OF DIGITS YOU WANT ( UP TO 4)"
25 INPUT L
30 X = EXP10(L) : R = X - 1
40 FOR I = 1 TO 100
50      PRINT INT(RND(0) * R),
60 NEXT I
70 PRINT: GOTO 10
```

-- STATEMENT --

EXT
Define Address of External Program

```
  EXT subname=address
     'subname' is a 1-6 character name for the external
       subroutine
     'address' is the memory address, in hexadecimal
       or integer notation, where the external subroutine
 originates.
```

You may interface an external object code program with your BASIC program by using EXT. EXT names the external subroutine and defines the memory address where the subroutine originates. To call the routine, use CALL.

Note: See the chapter on Segmenting Programs.

Examples

```
   EXT SUBPROG=&E000
```

the external routine named SUBPROG originates at the memory address of hex E000.

Sample Program

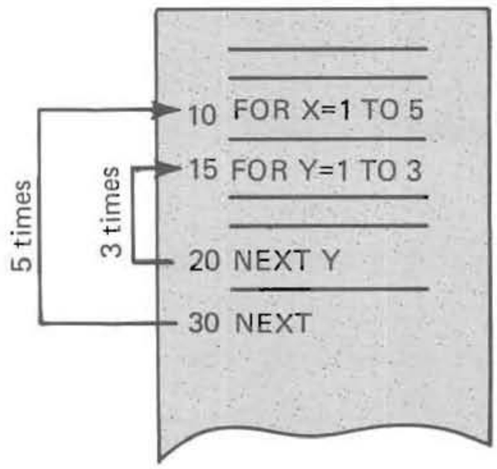
See the chapter on Segmenting Programs.

-- STATEMENT --

FOR/NEXT
Establish Program Loop

```
FOR variable = initial value TO final value STEP
increment
   'variable' is any numeric variable name;
      'variable' is optional after NEXT
   'initial value', 'final value', and 'increment'
      are numeric constants, variables, or
      expressions.
   STEP 'increment' is optional; if STEP 'increment'
      is omitted, a value of 1 is assumed.
```

FOR...TO...STEP/NEXT opens a repetitive loop so that a sequence of program statements may be executed over and over a specified number of times.



When BASIC executes the FOR statement for the first time, it sets the 'variable' to 'initial value'. Then 'variable' is compared with 'final value'. If 'variable' is greater than 'final value', BASIC completes the loop and goes to the statement following NEXT. (if 'increment' is a negative number, the loop ends when 'variable' is LESS than 'final value'.)

If 'variable' has not yet exceeded 'final value' BASIC continues executing the next statements until it encounters NEXT. At this point, BASIC goes back to FOR and increments the 'variable' by the amount specified in step 'increment'. (If 'increment' has a negative value, the 'variable' is actually decremented). STEP 'increment' is often omitted, in which case BASIC uses 1 as an increment. BASIC then repeats the whole process, Comparing 'variable' with 'final value'.

Examples

```
   FOR X = 1 TO 3
```

Sets up a loop which will be repeated 3 times: when X is 1, 2, and 3. (Since no STEP increment is specified, an increment of 1 is used).

This loop is closed by the following statement:

```
   NEXT X
```

```
   FOR I = 2 TO 6 STEP 2
```

Sets up a loop to be repeated 3 times: when I is 2, 4, and 6.

```
   FOR I = 8 TO 5 STEP -1
```

Sets up a loop to be repeated 4 times: when I is 8, 7, 6, and 5.

Both of the loops above are closed by the statement:

```
   NEXT I
```

Sample Programs

```
   80 REM      *** SAMPLE PROGRAM #1 DEMONSTRATING FOR/NEXT ***
   90 REM
  100 FOR I = 10 TO 1 STEP -1
  110        PRINT I;
  120 NEXT I
```

```
   80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING FOR/NEXT ***
   90 REM
  100 FOR I = 1 TO 3
  110 PRINT "OUTER LOOP"
  120      FOR J = 1 TO 2
  130           PRINT "      INNER LOOP"
  140      NEXT J
  150 NEXT I
*RUN
OUTER LOOP
      INNER LOOP
      INNER LOOP
OUTER LOOP
      INNER LOOP
      INNER LOOP
OUTER LOOP
      INNER LOOP
      INNER LOOP

STOP LINE 150
*SY "SCREEN"
```

-- STATEMENT --

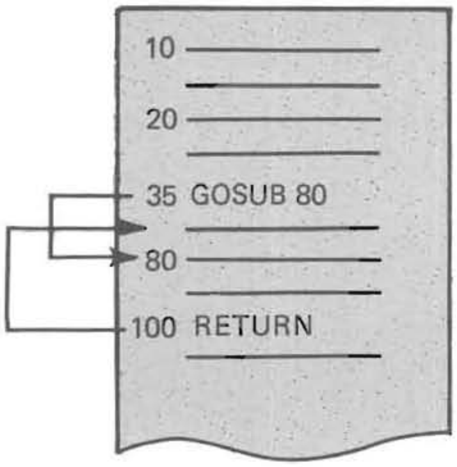
GOSUB
Go to Specified Subroutine

```
GO SUB line number
GOSUB line number
```

GO SUB or GOSUB (the space is optional) transfers program control to the subroutine beginning at the specified line number. Like GOTO, GOSUB is an unconditional or automatic program branch which may be conditional if it follows a test statement.

RETURN ends the subroutine by sending program control back to the line immediately following the GOSUB statement. All subroutines are ended by a RETURN statement.

NOTE: See also RETURN.



Examples

```
   GOSUB 1000
```

When this line is executed, control will automatically branch to the subroutine at 1000.

```
   IF A$ = "YES" THEN GOSUB 2000
```

Here, GOSUB is a conditional branch. If the condition is true, then control will branch to the subroutine at line 2000. However, if the condition is false, the program will immediately advance to the next line. GOSUB 2000 will be ignored.

Sample Program

```
   80 REM      *** SAMPLE PROGRAM DEMONSTRATING GOSUB ***
   90 REM
  100 GOSUB 120
  110 PRINT "BACK FROM THE SUBROUTINE" : STOP
  120 PRINT "EXECUTING THE SUBROUTINE"
  130 RETURN
*RUN
EXECUTING THE SUBROUTINE
BACK FROM THE SUBROUTINE
```

-- STATEMENT --

GOTO
Go To Specified Line Number

```
GO TO line number
GOTO line number
```

GO TO or GOTO (the space is optional) transfers program control to the specified line number. Used alone, GOTO results in an unconditional or automatic branch. However, a test may precede the GOTO to effect a conditional branch.

Examples

```
GOTO 100
```

When this line is executed, control will automatically be transferred to line 100.

```
IF A = 1 THEN PRINT "CORRECT": GOTO 50
```

In this statement, GOTO is used as a conditional branch. If A = 1, the Computer will print "CORRECT" and transfer control to line 50. However if A does not equal 1, control will drop to the next program line. GOTO 50 will be ignored.

Sample Program

```
10 REM     *** SAMPLE PROGRAM DEMONSTRATING GOTO ***
20 GOTO 40
25 PRINT "LINE 25"
27 STOP
30 PRINT "LINE 30"
35 GOTO 25
40 PRINT "LINE 40"
50 GOTO 30
```

-- FUNCTION --

HEX$
Compute Hexadecimal Value

```
HEX$(number)
   'number' is a numeric expression in the range
      -32768 to 32767.
```

HEX$ is the inverse of the HVL function. It returns a string which represents the hexadecimal value of the 'number'. Since the hexadecimal value is returned as a string, it cannot be used in a numeric expression. You cannot add, subtract, multiply or divide hex strings. You can concatenate them, though.

The hexadecimal string returned represents the value of the stored 'number'. Since the 'number' is an integer, it is stored in two's complement notation. HEX$(-1) returns the hexadecimal string "FFFF", since this is the way -1 is stored in two's complement notation. An explanation on the storage of integers is in the Programmers Information Section.

Examples

```
PRINT HEX$(30), HEX$(50), HEX$(90)
```

Prints the following strings:

```
001E            0032            005A
```

```
PRINT HEX$(-1), HEX$(-16), HEX$(-32768)
```

Prints the following strings:

```
FFFF            FFF0            8000
```

```
Y$ = HEX(X/16)
```

Y$ is the hexadecimal string representing the integer quotient

X/16.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING HEX$ ***
 90 REM
100 PRINT "INPUT A DECIMAL NUMBER FROM 1 TO 32767"
110 INPUT DEC
120 PRINT "HEXADECIMAL VALUE IS "; HEX$(DEC)
130 GOTO 100
```

-- FUNCTION --

HVL
Convert Hexadecimal String

```
   HVL(string)
      'string' is a string constant or a string variable.
```

HVL is the inverse of the HEX$ function. It returns the integer value of a hexadecimal string. Since integers are stored in two's complement notation, hexadecimal values over 7FFF will return negative integers.

NOTE: An explanation on the Storage of Integers is included in the Programmers Information Section

Examples

```
   PRINT HVL("7FFF")
```

Prints 32767.

```
   PRINT HVL("8000")
```

Prints -32768.

```
   PRINT HVL("4C IS THE CODE FOR L")
```

Prints 76. (HVL read the hexadecimal number "4C" and then stopped its search since the next character was not a hexadecimal character).

```
   H = HVL("F")
```

Assigns the value 15 to H.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING HVL ***
 90 REM
100 PRINT "TYPE A HEXADECIMAL NUMBER"
110 INPUT A$
120 N = HVL(A$)
130 IF N < 0 THEN D = N + 65536 ELSE D = N
140 PRINT "THE INTEGER REPRESENTATION FOR "; A$; " IS "; N
150 PRINT
160 PRINT A$; " CONVERTED TO A DECIMAL NUMBER IS"; D
170 PRINT
180 GOTO 100
```

-- STATEMENT --

IF...THEN...ELSE
Test Conditional Expression

```
IF test THEN statement or line number ELSE statement or
line number
   'test' is one or more relations connected by logical
      operators
      'relation' is two numeric or two string
         expressions separated by a relational
         operator
   'statement' is one or more BASIC statements
      separated by colons.  A line number may
      be substituted for 'statement'.
  ELSE statement is optional

  Note that 'statement' must be executable, e.g.,
  not a REM or DIM statement.
```

IF...THEN...ELSE tests the 'relation' to see if it is true. If it is true and there is more than one relation separated by logical operators, BASIC will continue testing each relational and logical operation in the statement.

If the 'test' returns a true result, the statement or statements following THEN will be executed. If the test returns a false result, control will jump to the statement or statements following ELSE, or, if ELSE is omitted, to the next program line.

The conditional statement GOTO 50 may be replaced by simply a line number.

Examples

```
   IF X > 127 THEN PRINT "OUT OF RANGE" : STOP
```

If X is greater than 127, the statement will be printed and program execution will stop. If X is not greater than 127, control will jump down to the next program line, skipping the

PRINT and STOP statements.

```
   IF X > 0 AND Y <> 0 THEN Y = X + 180
```

If both expressions are true, then Y will be assigned the value X + 180. Otherwise, control will pass directly to the next program line, skipping the THEN clause.

```
   IF A < B THEN PRINT "A < B" ELSE PRINT "B <= A"
```

If A is less than B the Computer prints the fact and then proceeds down to the next program line, skipping the ELSE statement. If A is not less than B, the Computer jumps directly to the ELSE statement and prints the "B <= A". Then control passes to the next statement in the program.

```
   IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN 400 ELSE 370.
```

If A$ is YES then the program branches to line 210. If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO then the program branches to line 400. If A$ is any value besides NO or YES, the program skips to the second ELSE and the program branches to line 370.

```
   IF A > .001 THEN B = 1/A : A = A/5 : ELSE 1510
```

If the value of A is greater than .001, then the next two statements will be executed, assigning new values to B and A. Then the program will drop down to the next line, skipping the ELSE statement. But if A is less than or equal to .001, then the program jumps directly over to ELSE, which then instructs it to branch to 1510. Note that GOTO is not required after ELSE.

Sample Programs

```
 80 REM     *** SAMPLE PROGRAM #1 DEMONSTRATING IF/THEN ***
 90 REM
100 PRINT "INPUT THE NUMBER 0 OR 1"
110 INPUT N
120 IF N=0 OR N=1 THEN STOP ELSE PRINT "NOT A BINARY DIGIT"
```

```
 80 REM      *** SAMPLE PROGRAM #2 DEMONSTRATING IF/THEN ***
 90 REM
100 PRINT "DO YOU WANT TO TEST THE IF/THEN STATEMENT"
110 INPUT A$
120 IF A$ = "YES" THEN PRINT "YOU INPUT YES": GOTO 100: ELSE IF
    A$ = "NO" THEN STOP ELSE PRINT "INPUT YES OR NO" : GOTO 110
```

```
10 REM     *** IF...THEN...ELSE STATEMENT ***
20 INPUT PROMPT="YES OR NO (Y/N)? "; R$
30 IF R$ = "Y" THEN 40 ELSE IF R$="N" THEN 50
35 GOTO 20
40 PRINT "THAT'S BEING POSITIVE!"
50 PRINT "WHY SO NEGATIVE?"
```

-- FUNCTION --

INKEY$
Get Keyboard Character if Available

```
INKEY$
```

Returns a one-character string from the keyboard without the necessity of having to press ENTER. If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed to the Display.

Example

```
A$ = INKEY$
```

When put into a loop, the above program fragment will get a key from the keyboard and store it in A$. If the line above is used by itself, when control reaches it and no key is being pressed, a null string ("") will be stored in A$.

Sample Programs

```
 10 REM    *** INKEY$ FUNCTION ****
 20 DIM C$1
 30 PRINT CHR$(27)
 40 PRINT "ECHO PROGRAM - TYPE ANY TEXT KEY AND IT WILL BE ECHOED"
 50 A$ = INKEY$
 60 IF A$ = "" THEN 50
 65 IF A$ < " " THEN 90
 70 PRINT A$;
 80 GOTO 50
 90 PRINT CHR$(26)
100 PRINT "CONTROL CHARACTERS ARE IGNORED - PRESS <BREAK> TO QUIT"
110 PRINT CHR$(25);
120 GOTO 50
```

```
ECHO PROGRAM - TYPE ANY TEXT KEY AND IT WILL BE ECHOED
O
CONTROL CHARACTERS ARE IGNORED - PRESS <BREAK> TO QUIT
```

-- STATEMENT --

INPUT
Input Data

```
  INPUT LENGTH=number, PROMPT=string; variable-list
    'string' is a string constant or a string variable.
        PROMPT=string; may be omitted.
    'variable-list' is a list of variables, with a comma
        after each but the last. The variable-types
        (string, integer, real) should match the data
        to be input.
    'number' is an integer value 1-255 specifying the
        maximum number of charactes to input.  If omitted,
        default is 255.
        LENGTH=number is optional.
```

This statement inputs data from the keyboard.

When executed, INPUT displays a question mark followed by an input buffer composed of 'number' dots. When you press <ENTER> or fill the input buffer, INPUT edits the input stream until it satisfies the input 'variable-list'. If the expected number of data items are found, INPUT is complete. If more are needed, INPUT displays another input buffer composed of 'number' dots, and waits for further input.

Special Keys During INPUT

<ENTER>	Ends the line at the current cursor position.
<ESC>	Erases the line and starts over.
<SPACEBAR>	Advances the cursor and types a blank space.
<BKSP>	Backspaces the cursor and erases character.
<-	Moves the cursor back without erasing.
->	Moves the cursor forward without typing a space.
<CTRL W>	Erases to the end of line.
<CTRL Z>	Erases to the end of input buffer.
<BREAK>	Cancels input. The <BREAK> is not recognized until you press <ENTER> or fill the input buffer.

All other keys are accepted as data for the input line. Control keys are echoed as '+/-', but are input correctly.

Examples

```
    INPUT A, B, C, D
```

Inputs values for the four variables listed.

```
    INPUT A$
```

Inputs a string value for A$

Sample Program

```
    10 REM       *** INPUT STATEMENT ***
    20 DIM NAME$25
    30 PRINT "ENTER DATA LIKE THIS: name, age"
    40 INPUT NAME$, AGE%
    50 PRINT: PRINT "HERE'S HOW THE DATA WAS EVALUATED:"
    60 PRINT "NAME: '"; NAME$; "'"
    70 PRINT "AGE:  '"; AGE%; "'"
    80 PRINT
    90 GOTO 30
```

Input Stream Edit Process

Leading spaces are always ignored. Beyond that, the editing process used depends on whether the target variable is string or numeric.

String Input

The string field starts with the first non-space character, and ends when a comma or carriage return is encountered. If a comma is encountered before any non-space characters, the target variable is given the null-string value, and input continues with the next target variable (if any). If a carriage return is encountered before any non-space characters, INPUT displays a new input buffer and waits for more data for the same target variable.

There is a special case when the first non-space character is a double-quote '"'. This causes all subsequent characters, including commas, to be accepted into the string, up to the next un-paired quote or carriage return (<ENTER>).

To include a double-quote in a quoted string, use paired double-quotes.

For example, the table below describes the result of the statement

```
    INPUT X$
```

under various conditions (<ENTER> represents a carriage return; "~" represents a leading or trailing blank space and is used only where necessary for illustration or emphasis).

```
Data stream                        Result in X$
-----------                        ------------

J.D. POWERS <ENTER>                'J.D. POWERS'
~~~J.D. POWERS~~~,                 'J.D. POWERS   '
FIRST, SECOND, THIRD <ENTER>       'FIRST'
, FIRST <ENTER>                    ''  (null string)
HE SAID "HI" <ENTER>               'HE SAID "HI"'
HE SAID "HI, JACK" <ENTER>         'HE SAID "HI'
"   J.D. POWERS   " <ENTER>        '   J.D. POWERS   '
"HE SAID ""HI""" <ENTER>           'HE SAID "HI"'
"HE SAID, ""HI, JACK."""           'HE SAID, "HI, JACK."'
```

Numeric Input

The numeric field starts with the first non-space character, and ends when a comma or carriage return is encountered. If the comma is encountered first, the target variable is given a value of zero, and input continues with the next target variable, if any. If a carriage return is first, INPUT displays a new 255-character input buffer and waits for more data for the same target variable.

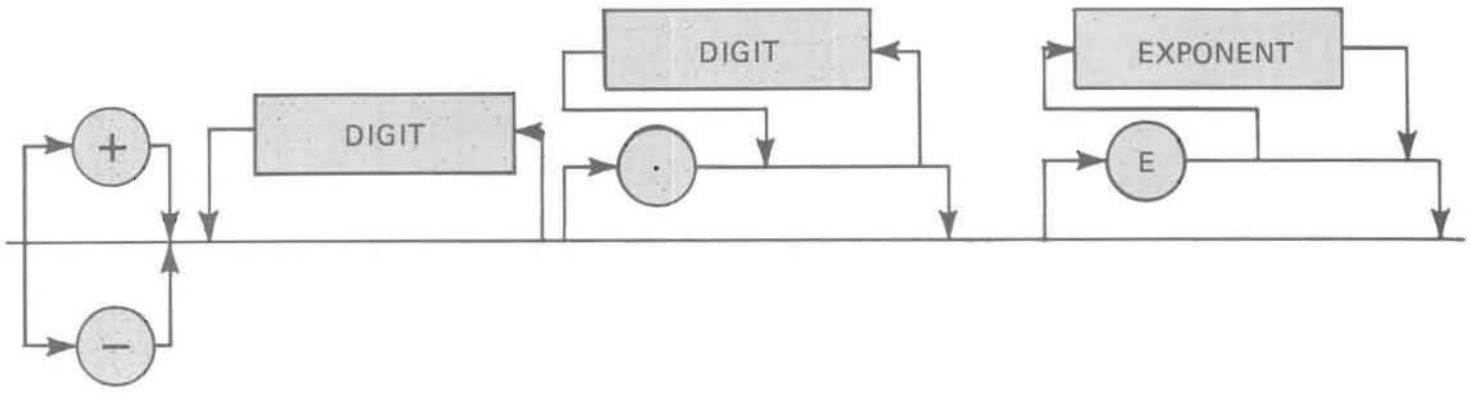
Once a numeric field has been delimited, INPUT evaluates the field. The following characters are valid in a numeric field:

    Digits 0-9
    Decimal point
    E (Exponent suffix)
    + and - signs
    Blank spaces (They are ignored.)

All other characters are invalid.

If an invalid character is encountered, input stops. The target variable receives the value of the field up to that point, and an error (INPUT SYNTAX ERROR #5) is generated.

Even valid characters may terminate a field, if they are used out of context. The following diagram shows the general form for a numeric field in which all the elements are valid (note that spaces may separate any two elements without having any effect on the evaluation):



'digit' is one of the characters from 0 through 9.
'exponent' is a whole number from -64 to +63. The sign is optional for positive values.

For example, the table below describes result of the statement

INPUT X$
under various conditions (<ENTER> represents a carriage return; "~" represents a leading or trailing blank space and is used only where necessary for illustration or emphasis).

Data stream	Result in X$
~~100~~~ <ENTER>	100
1 2 3 4 5,	12345

, 1 2 3 4 5 <ENTER>	0
-1.2345 E5 <ENTER>	-123450
+123450. E-5 <ENTER>	1.2345
100H <ENTER>	100 (Error #5)
1234/ <ENTER>	1234 (Error #5)
1..2 <ENTER>	1
..1 <ENTER>	0

```
    10 REM*** INPUT STATEMENT ***
    20 DIM MSG$80
    30 INPUT PROMPT="TYPE IN A MESSAGE: "; MSG$
    40 INPUT PROMPT="TYPE IN THREE NUMBERS: "; N1, N2, N3
    50 PRINT "DATA IS STORED LIKE THIS"
    60 PRINT "'"; MSG$; "'"
    70 PRINT N1, N2, N3
    80 PRINT: GOTO 30
 *RUN
 TYPE IN A MESSAGE: THIS IS A MESSAGE.
 TYPE IN THREE NUMBERS: 3, 12, 4
 DATA IS STORED LIKE THIS
 'THIS IS A MESSAGE.'
  3              12               4
```

-- STATEMENT --

INPUT from a disk file
Input Data From Disk File

Sequential access:
```
  INPUT # file-unit; variable-list
```

Indexed sequential access:
```
  INPUT # file-unit, KEY = key; variable-list
```

Direct access:
```
  INPUT # file-unit, KEY = record-number; variable-list

  'file-unit'  is a numeric expression specifying the
      output file. The file-unit number is assigned when
      the file is opened.
  'variable-list'  specifies the target variables to
      receive the data input from the file. Every
      variable but the last must be followed by a
      comma. There should  be no punctuation
      after the last variable.
  'KEY=key'  is used for input from indexed sequential
      access files. 'key' is a string expression
      containing the sort key.
  'KEY=record-number'  is used for input from direct
      access files. 'record-number' is a numeric
      expression specifying the record number.
```

This statement inputs data from a disk file. The data should have been written by an analogous PRINT to disk file statement. The number and type of target variables should match the number and type of values in the PRINT item-list.

The input stream edit process is like that of INPUT from the keyboard.

Examples

```
    INPUT #1; A, B, C, D
```

Inputs values for A, B, C and D from file-unit #1.

```
    INPUT #2, KEY=NAME$; PAYRAT, EXEMPT%
```

Inputs values for PAYRAT and EXEMPT% from the record indexed by the contents of NAME$, from file-unit #2.

```
    INPUT #3, KEY=RECORD%; PAYRAT, EXEMPT%
```

Inputs values for PAYRAT and EXEMPT% from the direct-access record specified by RECORD%, from file-unit #3.

Sample Program

See the chapter on data files.

-- STATEMENT --

INPUT USING
Input Formatted Data

```
INPUT USING LENGTH=number, PROMPT=string; variable-list
   'string' is a string constant or string variable.
       PROMPT=string; may be omitted.
   'image' specifies the format of the data; it
       can be a line number referring to an image
       statement, or a string constant or string
       variable containing the image specifiers.
   'variable-list' is a list of one or more variables,
       with a comma after each but the last. The
       variable-types (string, integer, real)
       should match the data to be input.
   'number' is an integer specifying the maximun number
       of characters to input.
       LENGTH=number is optional.  The default value is
       255.
```

INPUT USING inputs data from the keyboard according to a specified format--how many fields, how many characters in each field, and which characters to skip over.

You specify the format with an image line--either contained on a separate program-line, or in a string variable referenced in the INPUT USING statement. Image lines contain special characters indicating the positions and lengths of fields within the data.

When executed, INPUT USING displays a question mark followed by an input buffer composed of 'number' dots. When you press <ENTER> or fill the input buffer, INPUT USING edits the data until it finds enough fields to satisfy the input 'variable-list'. If the expected number of data fields are not found, INPUT USING displays a new input buffer and waits for more data.

Special Keys During INPUT USING

<ENTER>	Terminates the line at the current cursor position and begins input-stream editing.
<ESC>	Erases the line and starts over.
<SPACEBAR>	Advances the cursor and types a blank space.
<BKSP>	Backspaces the cursor and erases character.

<-	Moves the cursor back without erasing.
->	Moves the cursor forward without typing a blank space.
<CTRL W>	Erases to the end of line.
<CTRL Z>	Erases to the end of input buffer.
<BREAK>	Cancels input. The <BREAK> is not recognized until you press <ENTER> or fill the input buffer.

All other keys are accepted as data for the input line. Control keys are echoed as '+/-', but are input correctly.

Image Lines for INPUT USING

If stored in a separate program line, image lines take this form:

```
 nnnnnb;image
   'nnnnnb' is the line number, followed by a blank space
   ';' marks the line as a non-executable image line
   'image' is a sequence of characters defining the image
      format, as follows:
      '#' specifies a numeric or string character.
         A sequence of N "#" characters represents a
    numeric or string field of N characters.
```

You can also store the image inside a string variable. Simply assign the appropriate image character sequence to the string variable.

Examples

```
    100 IMAGE$ = "########## ##### ####### #####"
    110 INPUT USING IMAGE$, FIELD1$, FIELD$, FIELD3,
FIELD4%
```

Inputs values for the four variables listed, using the image contained in IMAGE$.

```
    100 ;#######
    110 INPUT USING 100, RATE
```

Inputs a value for RATE, according to the image statement in line 100.

Sample Programs

```
100 REM            *** INPUT USING ***
110 DIM NAME$25, IMAGE$28
120 REM        :---25 character name---: nn
130 IMAGE$ = "######################### ##"
140 PRINT "TYPE IN A LINE LIKE THIS (name, age)"
150 PRINT TAB(2); IMAGE$
160 INPUT USING IMAGE$, NAME$, AGE%
170 PRINT: PRINT "DATA WAS EVALUATED LIKE THIS:"
180 PRINT "NAME: '"; NAME$; "'"
190 PRINT "AGE:  '"; AGE%;  "'"
200 PRINT: GOTO 140
```

The following program uses a separate image line:

```
100 PRINT "ENTER A NUMBER (UP TO 10 DIGITS)"
110 INPUT USING 120, A
120 ;##########
130 PRINT "THE DATA WAS EVALUATED LIKE THIS:"
140 PRINT USING 120, A
150 GOTO 100
```

When you run the program, always input 10-digit numbers (including sign, decimal point, exponent field, etc.). Otherwise, the data evaluation will probably differ from what you intended. For further details, read "INPUT USING Edit Process."

INPUT USING Edit Process

The 'image' defines the fields which are passed to the standard input evaluation routines. The image serves as a "mask", in that only those characters aligned with "#" signs are used. For example:

```
Image:             "########## #####"
Data:              "MR. JONES   1.334567"
Resultant fields: "MR. JONESb" and "bl.33"
```

("b" represents a blank space and is used only where necessary for purposes of illustration or emphasis.)

String Input

All characters in the field are input to the target variable--including leading and trailing spaces, commas and quotes. There are no special delimiters.

For example, the table below describes result of the statement

```
   INPUT USING A$, S1$, S2$
```

under various conditions ("~" represents a leading or trailing blank space and is used only where necessary for illustration or emphasis).

A$ (Image)	Data	Result S1$	Result S2$
# ########	ABCDEFGHIJK	A	CDEFGHIJ
## #######	ABCDEFGHIJK	AB	DEFGHIJ
#### #####	G-44 L-5	G-44	L-5bb
#### #####	A,B,C,D,E	A,B,	,D,E,
# #~~~~	FIRST SECOND	F~~~~	S~~~~

Numeric Input

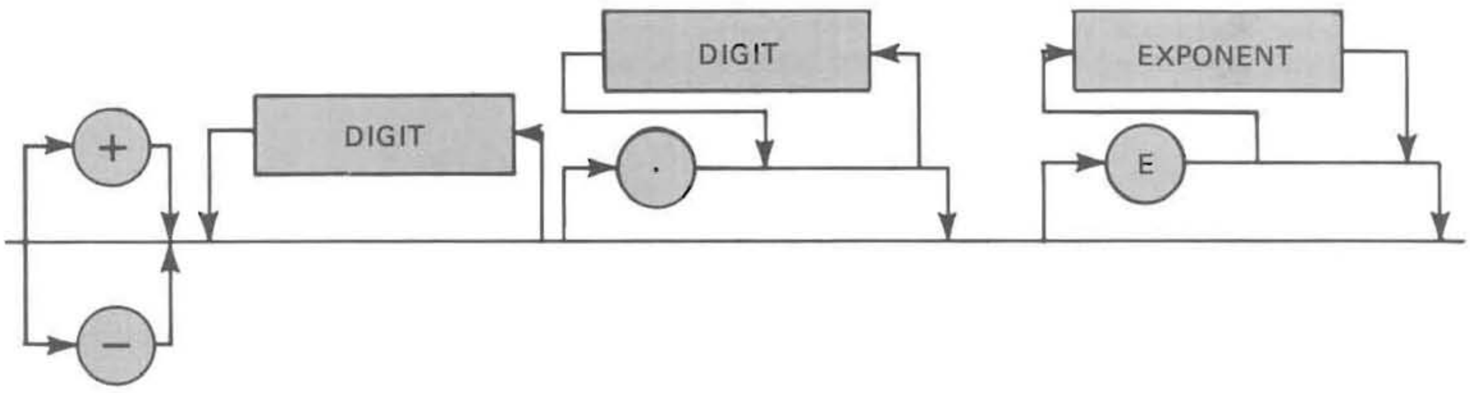
The following characters are valid in a numeric field:

Digits 0-9
Decimal point
E (Exponent suffix)
+ and - signs
Blank spaces (They are interpreted as zeroes.)

If a comma is encountered in the input data, evaluation stops and the current target variable receives the value of the field up to that point. If there are additional target variables to be filled, INPUT USING continues evaluation of the input line. The evaluation continues at the first character following the current image field.

All other characters are invalid. If an invalid character is encountered, input stops. The target variable receives the value of the field up to that point, and an error (INPUT SYNTAX ERROR #5) is generated.

Even valid characters may terminate a field, if they are used out of context. The following diagram shows the general form for a numeric field in which all the elements are valid (note that spaces may separate any two elements without having any effect on the evaluation):



'digit' is one of the characters from 0 through 9.
'exponent' is a whole number from -64 to +63. The sign is optional for positive values.

For example, the table below describes result of the statement

```
   INPUT USING A$, S1, S2%
```

under various conditions ("~" represents a leading or trailing blank space and is used only where necessary for illustration or emphasis).

		Result	
A$ (Image)	Data	S1	S2%
##### ####	1234567890	12345.	7890
##### ###	~~~10 12	10.	12
######## #	-1.234E5 1	123400.	1
##### ####	100, 2000	100.	2000
##### ####	100,2000	100.	0*
#####.####	12345.67890	12345.	6789
###### #	1 1	10000.	1

* Zero because the '2' after ',' is forced into alignment with the blank space in the image. Compare with the preceding line in the table.

-- STATEMENT --

INPUT USING from a disk file
Input Formatted Data From Disk File

```
  Sequential access:
    INPUT USING # file-unit; image, variable-list

  Indexed sequential access:
    INPUT USING # file-unit, KEY = key; image, variable-list

  Direct access:
    INPUT USING # file-unit, KEY = record-number; image,
variable-list

    'file-unit'  is a numeric expression specifying the
        output file. The file-unit number is assigned when
        the file is opened.
    'image'  specifies the format of the data; it can be a
        line number referring to an image statement, or a
        string expression containing the image.
    'variable-list'  specifies the target variables to
        receive the data input from the file. Every
        variable but the last must be followed by a
        comma. There should  be no punctuation
        after the last variable.
    'KEY=key'  is used for input from indexed sequential
        access files. 'key' is a string expression
        containing the sort key.
    'KEY=record-number'  is used for input from darect
        access files. 'record-number' is a numeric
        expression specifying the record number.
```

This statement inputs formatted data from a disk file in a manner analogous to INPUT USING from the keyboard. The data should have been written by an analogous PRINT to disk file statement. The number and type of target variables should match the number and type of values in the PRINT item-list.

For further details on image specifiers and input stream editing, see INPUT USING from the Keyboard.

Examples

```
    INPUT USING #1; "####### ## ####### #####", A, B, C, D
```

Inputs values for A, B, C and D using the indicated image, from file-unit #1.

```
    INPUT USING #2, KEY=NAME$; FMT$, PAYRAT, EXEMPT%
```

Inputs values for PAYRAT and EXEMPT% from the record indexed by the contents of NAME$, using the image in FMT$, from file-unit #2.

```
    100 ;####### ##

    200 INPUT USING #3, KEY=RECORD%; 100, PAYRAT, EXEMPT%
```

Inputs values for PAYRAT and EXEMPT% from the direct-access record specified by RECORD%, using the image in line 100, from file-unit #3.

Sample Program

See the chapter on data files.

-- FUNCTION --

INPUT$
Input a Character String

```
INPUT$(length)
    'length' is a numeric expression in the range
       of 1 to 255.
```

INPUT$ causes the program to stop execution until the operator inputs a string with the 'length' specified. For example, INPUT$(3) causes the program to stop until the operator inputs 3 characters. Once the operator inputs these 3 characters, the program immediately resumes execution.

The operator can input less than the 'length' required by pressing <ENTER> after completing the input.

Examples

```
   A$ = INPUT$(5)
```

The program stops until the operator presses either 5 characters, or less than 5 characters followed by <ENTER>. This string is assigned to A$.

```
   IF INPUT$(3) = "YES" THEN 500
```

The program stops until the operator presses 3 characters (or less than 3 followed by <ENTER>). After the 3rd character is pressed, the Computer executes the rest of the IF/THEN statement.

```
   LPRINT INPUT$(20)
```

At this line, the program stops to allow the operator to input a maximum of 20 characters. These characters are then printed on the line printer.

Sample Program

```
    80 REM      *** SAMPLE PROGRAM DEMONSTRATING INPUT$ ***
    90 REM
   100 REM      *** MAILING LIST -- LAST TWO ENTRIES ***
   110 REM
   120 PRINT "TYPE THE STATE -- MUST BE 2 CHARACTERS"
   130 A$ = INPUT$(2)
   140 PRINT "TYPE THE ZIP CODE -- MUST BE 5 CHARACTERS"
   150 B$ = INPUT$(5)
   160 ADDRESS$ = A$ & " " & B$ : PRINT ADDRESS$
 *RUN
 TYPE THE STATE -- MUST BE 2 CHARACTERS
 TX
 TYPE THE ZIP CODE -- MUST BE 5 CHARACTERS
 76118
 TX 76118
```

-- FUNCTION --

INT
CONVERT TO INTEGER VALUE

```
INT(number)
   'number' is any numeric expression.
```

INT returns the largest whole number that is not greater the the 'number'. Unlike CVI, the number is NOT limited to the range [-32768, 32767].

Examples

```
   A = INT(X)
```

Gets the integer value of X and stores it in A.

```
   PRINT INT(2.5)
```

Prints 2.

```
   PRINT INT(-2.5)
```

Prints -3.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING INT ***
 90 REM
100 PRINT "ENTER A 6-DIGIT POSITIVE NUMBER LIKE XX.XXXX"
110 INPUT X
120 IF X<0 THEN 100
130 A = INT((X * 100) + 0.5) / 100
140 PRINT X; "ROUNDED TO TWO DECIMAL PLACES IS"; A
150 GOTO 100
```

-- STATEMENT --

INTEGER
Define Variables as Integers

```
INTEGER*2 letter list
  *2 represents the 2-byte length of the integers.
    This may be omitted.
  'letter list' is a sequence of individual letters
    or letter ranges; the elements of the list must
    be separated by commas.  A letter range is in the
    form:
       letter1-letter2
    If omitted, all variables will be defined as
    integers.
```

Ordinarily, BASIC classifies all variables as real unless a definition statement or type declaration tag tells it to do otherwise. INTEGER changes this default from real to integer.

If a 'letter list' is used, only variable names beginning with the letters specified will be defaulted. Integer values must be in the range of -32768 to 32767. They are stored internally in two-byte, two's complement form.

INTEGER cannot be used after an executable statement.

Note: For more information, see the chapter on BASIC Concepts.

Examples

```
   INTEGER A, I, N
```

After the above line, all variables beginning with A, I, or N will be treated as integers. For example, A1, AA, and I3 will be integer variables. However, A1$, AA$, and I3$ would still be string variables, because the type-declaration characters always override the INTEGER statement.

```
   INTEGER I-N
```

Causes any variable beginning with the letters I through N to be treated as integer variable.

INTEGER

All variables in the program will be treated as integers unless they have a type declaration tag, or there is a STRING or REAL statement following this.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING INTEGER ***
 90 REM
100 INTEGER W
110 Z = 1.9 : W = 1.9
120 PRINT "THE VALUE OF REAL NUMBER Z IS "; Z
130 PRINT "BUT THE VALUE OF INTEGER W IS "; W
```

-- FUNCTION --

KEY$
Returns key name.

```
KEY$
```

The KEY$ function returns the key name of a record in an indexed access file.

Example:

PRINT KEY$

prints the key of the current record.

Sample Program

```
10 OPEN #1, "LIST/DAT", MODE=R, TYPE=I, LENGTH=32, KEY=3
20 INPUT USING #1; 100, LNAME$, FNAME$, ADD$
30 IF EOF(#1) <> 0 THEN 80
40 PRINT
50 PRINT LNAME$;", "; FNAME$,,,,ADD$
60 PRINT KEY$
70 GOTO 20
80 CLOSE #1
100 ;<##########<######<################
```

-- STATEMENT --

KILL
Kill Disk File

```
KILL filespec
   'filespec' is a string constant or a string
       variable representing a TRSDOS file
       specification.  If it is a constant, it
       must be enclosed in quotes.
```

When the KILL statement is executed, the 'filespec' will be deleted from the disk directory. It may no longer be accessed and will be replaced by another file. KILL will not prompt you before deleting the file, so you might want to write a prompt as part of your program.

Examples

```
   KILL "FILE/BAS:1"
```

When this statement is executed, the file FILE/BAS from the disk in drive 1 will be deleted from the disk.

```
   KILL A$
```

The filespec stored as A$ is deleted from the disk.

Sample Program

```
 5 REM     *** SAMPLE PROGRAM DEMONSTRATING KILL ***
 6 REM
10 PRINT "INPUT THE FILE SPECIFICATION YOU WANT TO KILL"
15 PRINT "YOU WILL NOT BE PROMPTED -- "
17 PRINT "THE FILE WILL IMMEDIATELY BE DELETED"
18 PRINT "WITH NO WAY TO RECOVER IT"
20 INPUT A$
30 KILL A$
40 GOTO 10
```

-- FUNCTION --

LEN
Get Length of String

```
LEN(string)
  'string' is a string constant or a string variable.
```

LEN returns the current number of characters in the 'string'.

Examples

```
   PRINT LEN("MARY")
```

Prints 4.

```
   PRINT LEN("MARY HAD A")
```

Prints 10.

```
   X = LEN(SENTENCE$)
```

Stores the number of characters in SENTENCE$ in X.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING LEN ***
 90 REM
100 PRINT "INPUT WORDS OR A SHORT SENTENCE"
110 INPUT A$
120 PRINT "YOUR SENTENCE HAS"; LEN(A$); "CHARACTERS"
130 GOTO 100
```

-- STATEMENT --

LINE INPUT
Input Line of Data

```
LINE INPUT LENGTH=number, PROMPT=string; string variable
 The blank space in 'LINE INPUT' is optional.
 'string' is a string constant or a string variable.
     PROMPT=string; may be omitted.
 'string-variable' is the target variable
     for the input data.
  'number' is an integer value specifying the maximum
     number of characters to input.
     LENGTH=number, is optional.  If omitted, the
     default value of 255 is used.
```

When executed, LINE INPUT displays a question mark followed by an input buffer of 'number' dots. When you press <ENTER> or fill the input buffer, LINE INPUT accepts the line into the target variable.

Special Keys During INPUT

<ENTER>	Ends the line at the current cursor position.
<ESC>	Erases the line and starts over.
<SPACEBAR>	Advances the cursor and types a blank space.
<BKSP>	Backspaces the cursor and erases character.
<-	Moves the cursor back without erasing.
->	Moves the cursor forward without typing blank a space.
<CTRL W>	Erases to the end of line.
<CTRL Z>	Erases to the end of input buffer.
<BREAK>	Cancels input. The <BREAK> is not recognized until you press <ENTER> or fill the input buffer.

All other keys are accepted as data for the input line. Control keys are echoed as '+/-', but are input correctly.

Examples

```
   LINE INPUT TXT$
```
Inputs a line of characters into TXT$.

Sample Program

```
10 REM            *** LINE INPUT ***
20 DIM TXT$255
30 PRINT "TYPE IN A LINE OF TEXT--ANY CHARACTERS AT ALL"
40 LINE INPUT TXT$
50 PRINT "HERE'S HOW THE DATA IS SAVED"
60 PRINT "'"; TXT$; "'"
70 PRINT: GOTO 30
```

Input Stream Edit Process

Unlike INPUT, LINE INPUT does not ignore leading blanks. Every character you type (except the special keys listed previously) is accepted as data into the target variable. There are no invalid characters, and there are no terminators except for <ENTER>.

For example, the table below describes the result of the statement

```
   LINE INPUT USING X$
```

under various conditions (<ENTER> represents a carriage return; "~" represents a leading or trailing blank space and is used only where necessary for illustration or emphasis).

Data stream	Result in X$
J.D. POWERS <ENTER>	'J.D. POWERS'
~~J.D. POWERS~~~ <ENTER>	' J.D. POWERS '
FIRST, SECOND, THIRD <ENTER>	'FIRST, SECOND, THIRD'
HE SAID "HI" <ENTER>	'HE SAID "HI"'
HE SAID, "HI, JACK" <ENTER>	'HE SAID, "HI, JACK"'
TWO DOUBLE-QUOTES ""	'TWO DOUBLE-QUOTES ""'

```
10 REM  *** LINE INPUT  ***
20 DIM TXT$255
30 PRINT "TYPE IN A LINE OF TEXT--ANY CHARACTERS AT ALL":
40 LINE INPUT TXT$
50 PRINT "HERE'S HOW THE DATA IS SAVED"
60 PRINT "'"; TXT$; "'"
70 PRINT: GOTO 30
```

-- STATEMENT --

LINE INPUT from a disk file
Input Line of Data from Disk File

```
 Sequential access:
   LINE INPUT # file-unit; string-variable

 Indexed sequential access:
   LINE INPUT # file-unit, KEY = key; string-variable

 Direct access:
   LINE INPUT # file-unit, KEY = record-number;
string-variable

   The blank space in 'LINE INPUT' is optional.
   'file-unit'  is a numeric expression specifying the
       output file. The file-unit number is assigned when
       the file is opened.
   'string-variable'  is the target variables for
       the input data.
   'KEY=key'  is used for input from indexed sequential
       access files. 'key' is a string expression
       containing the sort key.
   'KEY=record-number'  is used for input from direct
       access files. 'record-number' is a numeric
       expression specifying the record number.
```

This statement inputs a line of data from a disk file and stores it in a string variable. For disk input, a line of data is terminated by any of the following:
. A carriage return
. Reception of 255 characters without a carriage return
. End of file

The input stream edit process is like that of LINE INPUT from the keyboard.

Examples

```
    LINE INPUT #1; A$
```

Inputs a value for A$ from file-unit #1.

```
   LINE INPUT #2, KEY=NAME$; COMMENTS$
```

Inputs a value for COMMENT$ from the record indexed by the contents of NAME$, from file-unit #2.

```
   LINE INPUT #3, KEY=RECORD%; COMMENT$
```

Inputs a value for COMMENT$ from the direct-access record specified by RECORD%, from file-unit #3.

Sample Program

See the chapter on data files.

-- FUNCTION --

LOG
Compute Natural Logarithm

```
LOG(number)
   'number' is a numeric expression.
```

LOG returns the natural logarithm of the 'number'. This is the inverse of the EXP function, so X = LOG(EXP(X)). To find the logarithm of a number to another base B, use the formula LOG B(X) = LOG E(X)/LOG E(B). For example, LOG(32767)/LOG(2) returns the logarithm to base 2 or 32767.

The result is always a real number.

Examples

```
    B = LOG(A)
```

Computes the value of LOG(A) and stores it in B.

```
   PRINT LOG(3.14159)
```

Prints the value 1.4473.

```
   Z = 10 * LOG(P2/P1)
```

Performs the indicated calculation and assigns it to Z.

Sample Program

```
10 PRINT "INPUT A NUMBER"
20 INPUT N
30 PRINT "THE NATURAL LOGARITHM OF"; N; "IS"; LOG(N)
40 GOTO 10
```

-- FUNCTION --

LOG10
Compute Base 10 Logarithm

```
LOG10(number)
  'number' is any numeric expression
```

LOG10 returns the base 10 logarithm of the 'number'. This is the inverse of the EXP10 function, so X=LOG10(EXP10(X)).

Examples

```
PRINT LOG10(100)
```

Prints 2.

```
X = LOG10(Y)
```

Assigns the value LOG10(Y) to X.

```
X = 10/LOG10(X + 2A)
```

Performs the calculation and assigns the results to X.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING LOG10 ***
 90 REM
100 PRINT "INPUT A NUMBER"
110 INPUT N
120 PRINT N; " = 10 TO THE POWER OF"; LOG10(N)
130 GOTO 100
```

-- STATEMENT --

LPRINT
Print on Line Printer

```
LPRINT item-list
  'item-list' contains expressions to be evaluated and
      output to the printer.  'item-list' may
      also contain TAB functions. Every item but the
      last must be followed by a semi-colon or comma.

      A semi-colon leaves the carriage in its current
      position; a comma advances the carriage to the next
      print zone.

      Unless a semi-colon or comma follows the last
      item, LPRINT will output a carriage return
      after the last character is displayed.
```

This statement outputs to the printer, beginning at the current carriage position. It works just like PRINT, except for those details specific to the video display.

Before using LPRINT, you must initialize the printer with the TRSDOS FORMS command. This establishes the line-width, page-length, and other parameters. See FORMS in the TRSDOS Reference Manual.

Control Codes

The following control codes are intercepted and handled by TRSDOS:

Code Hex.	Dec.	Action Taken
9	09	Tabs to next eight column boundary
0A	10	Ignored (not needed by Radio Shack line printers).
0C	12	Form feed.

All other codes are sent unchanged to the printer.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING LPRINT ***
 90 REM
100 REM      *** CHECK THAT LINE PRINTER IS CONNECTED AND ON-LINE ***
110 REM
120 PRINT "INPUT WHAT YOU WANT PRINTED ON THE LINE PRINTER"
130 INPUT A$
140 LPRINT A$
150 GOTO 120
```

-- STATEMENT --

LPRINT USING
Print Using Format on Line Printer

```
LPRINT USING image, item-list
  'image' specifies the format of the data; it can
      be a line number referring to an image
      statement, or a string expression containing
      the image specifiers.
  'print-function' is an optional use of TAB.
      If omitted, printing starts at the current
      carriage position.
  'item-list' contains expressions to be evaluated
      and output to the printer. TAB may be anywhere in
      the item list.  Every item but the last
      must be followed by a comma or semi-colon.
      However, a comma or semi-colon after the last
      item will suppress the automatic carriage return
      after the last character is printered. The
      carriage will remain in the next position
      following the last character printered.
```

This statement outputs to the printer, beginning at the current carriage location. Unlike LPRINT, it outputs formatted data, according to an image specification contained on a separate program line or in a string variable.

LPRINT USING is just like PRINT USING, except for the special features related to the video display.

Before using LPRINT, you must initialize the printer with the TRSDOS FORMS command. This establishes the line-width, page-length, and other parameters. See FORMS in the TRSDOS Reference Manual.

Control Codes

The following control codes are intercepted and handled by TRSDOS:

Code Hex.	Dec.	Action Taken
9	09	Tabs to next eight-column boundary.
0A	10	Ignored (not needed by Radio Shack line printers).
0C	12	Form feed.

All other codes are sent unchanged to the printer.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING LPRINT USING ***
 90 REM
100 TOTAL = 0
110 ; >######.##
120 ; >#########
130 FOR I = 1 TO 25
140     N = RND(0) * 99
150     LPRINT USING 110, N
160     TOTAL = TOTAL + N
170 NEXT I
180 LPRINT USING 120, "__________"
190 LPRINT USING 110, TOTAL
```

-- STATEMENT --

ON BREAK GOTO
Enable a <BREAK> Handling Routine

```
ON BREAK GOTO line number
```

Normally, when you hit the <BREAK> key while executing a program, BASIC stops your program and puts you in the command mode. You then must start your program at the beginning again.

You might want BASIC to handle the <BREAK> key in a different way. ON BREAK GOTO tells BASIC to go to the line number you specify whenever the <BREAK> key is pressed.

Note: See also RESET BREAK

Example

```
   ON BREAK GOTO 500
```

Whenever a <BREAK> key is pressed, control will go to line number 500.

Sample Program

```
 10 REM      **** ON BREAK GOTO AND RESET BREAK STATEMENTS ****
 20 PRINT CHR$(27);
 30 ON BREAK GOTO 160
 40 PRINT "I'M TRAPPING THE <BREAK> KEY NOW"
 50 PRINT "PRESS <BREAK> WHILE I COUNT TO 1000"
 60 FOR I = 1 TO 1000
 70   PRINT CRT(12,30), I
 80 NEXT I
 90 RESET BREAK
100 PRINT "NOW BREAK IS RESET"
110 PRINT "TRY PRESSING <BREAK> WHILE I COUNT TO 1000"
120 FOR I = 1 TO 1000
130   PRINT CRT(12,30); I
140 NEXT I
150 STOP
160 PRINT CHR$(27);  "YOU PRESSED <BREAK>"
170 GOTO 90
```

-- STATEMENT --

ON ERROR GOTO
Set Up Error-trapping Routine

```
ON ERROR GO TO line number
ON ERROR GOTO line number
```

ON ERROR GO TO or ON ERROR GOTO (the space is optional) allows you to set up an error-trapping routine to get the Computer to handle the error the way you want it handled. Normally, you have a particular error in mind when you use the ON ERROR GOTO statement.

This statement is often used to prevent error messages from confusing an operator who is a non-programmer. For example, if the operator inputs the wrong data type in any of your input statements, the Computer will break program execution and print an Input Syntax error message. To prevent this from happening you can set up an error trapping routine like the one demonstrated in the sample program.

The ON ERROR GOTO statement must be executed before the error occurs or it will have no effect. Once it has "trapped" an error, ON ERROR GOTO is disabled. You must use another ON ERROR GOTO statement to trap the next error.

A good way to use ON ERROR GOTO is to place it before any statement which might cause an error. If no error occurs, the next ON ERROR GOTO statement will supercede it.

NOTE: See also ERR, ERROR, and RESET ERROR

Example

```
   ON ERROR GOTO 1500
```

If an error occurs in your program anywhere after this line, control will branch to line 1500.

Sample Program

```
   80 REM      *** SAMPLE PROGRAM DEMONSTRATING ON ERROR GOTO ***
   90 REM
  100 ON ERROR GOTO 140
  110 PRINT "INPUT A WORD "
  120 INPUT A
  130 STOP
  140 IF ERR <> 5 THEN ERROR ERR
  150 PRINT "SORRY, YOU HAVE TO INPUT A NUMBER"
  160 REM
  170 REM      *** NEXT STATEMENT RE-ENABLES ON ERROR GOTO ***
  180 REM
  190 ON ERROR GOTO 140
  200 GOTO 120
*RUN
 INPUT A WORD
 ? WORD
 SORRY, YOU HAVE TO INPUT A NUMBER
 ? 67
```

-- STATEMENT --

ON...GOSUB
Test and Branch to Subroutine

```
ON test-value GOSUB line number, line number, ...
    'test-value' is a numeric expression.
```

ON...GO SUB or ON...GOSUB (the space is optional) is a multi-way branching statement like ON GOTO, except that control passes to a subroutine rather than just being shifted to another part of the program. For further information, see ON GOTO

Example

```
ON Y GOSUB 1000, 2000, 3000
```

This statement will first evaluate Y. If Y = 1, the subroutine beginning at line 1000 sill be called. If Y = 2, the subroutine at 2000 will be called. If Y = 3, the subroutine at line 3000 will be called.

Sample Program

```
   80 REM      *** SAMPLE PROGRAM DEMONSTRATING ON ... GOSUB ***
   90 REM
  100 PRINT "CHOOSE 1, 2, OR 3"
  110 INPUT I
  120 ON I GOSUB 500, 600, 700
  130 STOP
  500 PRINT "SUBROUTINE #1" : RETURN
  600 PRINT "SUBROUTINE #2" : RETURN
  700 PRINT "SUBROUTINE #3" : RETURN
*RUN
CHOOSE 1, 2, OR 3
? 3
SUBROUTINE #3
```

-- STATEMENT --

ON...GOTO
Test and Branch to Different Program Line

```
   ON test-value GOTO line number, line number, ...
      'test-value' is a numeric expression.
```

ON...GO TO or ON...GOTO (the space is optional) is a multi-way branching statement that is controlled by test value.

When the Computer executes ON GOTO, it first evaluates 'test-value' and, if it is a real number, converts it to an integer. We'll refer to this integer as J. The Computer then transfers control to the Jth line number in the ON GOTO statement. For example, if J = 1, the Computer transfers control to the first line number following GOTO; if J = 5, the program control drops to the fifth line number.

If 'test value' is smaller than one or greater than the number of line numbers in the list, the computer will proceed to the next program line.

Examples

```
   ON A GOTO 100, 200, 300
```

If the integer of A equals 1, program control drops to 100.
If it equals 2, program control drops to 200.
If it equals 3, program control drops to 300

```
   ON X GOTO 500, 520, 540, 550, 560
```

If integer A equals 1, program control drops to line 500.
If it equals 2, program control drops to line 520.
If it equals 3, program control drops to line 540.
If it equals 4, program control drops to line 550.
If it equals 5, program control drops to line 560.

Sample Program

```
   80 REM     *** SAMPLE PROGRAM DEMONSTRATING ON...GOTO ***
   90 REM
  100 PRINT "DO YOU WANT TO --- "
  110 PRINT " (1) INPUT FILES"
  120 PRINT " (2) REVISE FILES"
  130 PRINT " (3) LIST FILES"
  140 PRINT "INPUT 1, 2, OR 3"
  150 INPUT A
  160 ON A GOTO 500, 600, 700
  500 PRINT "INPUT FILES PROGRAM" : STOP
  600 PRINT "REVISE FILES PROGRAM" : STOP
  700 PRINT "LIST FILES PROGRAM" : STOP
*RUN
DO YOU WANT TO ---
 (1) INPUT FILES
 (2) REVISE FILES
 (3) LIST FILES
INPUT 1, 2, OR 3
? 2
REVISE FILES PROGRAM
```

-- STATEMENT --

OPEN
Open Disk File

```
OPEN #file-unit, file, MODE=m, TYPE=t, LENGTH=l, KEY=k

  'file-unit'  is a numeric expression; while the file is
      open, this number will be used to reference that
      file for disk I/O statements and functions.
  'file'  is a string expression containing a TRSDOS file
      specification for the file to be opened. If 'file'
      is a string constant, it must be enclosed in double
      quotes.
  'MODE=m'  specifies the access mode. 'm' is one mf the
      following:
      R   Read only
      E   Extend (i.e., sequential write beginning at the
          end of the file)
      U   Update (i.e., read or write to an existing
          direct or ISAM file)
      W   Write
  'TYPE=t'  specifies the file-type. 't' is one of the
      following:
      D, R Direct (random) access file (i.e., records are
          referenced by record number)
      I   Indexed sequential access file (ISAM, i.e.,
          records are referenced by a sorting key)
      S   Sequential (i.e., records are referenced in
          sequence)
  'LENGTH=l'  specifies the length of data in each
record.  (BASIC adds any necesssary overhead).
      'l' is a numeric expression with a value from 0 to 255.
      Sequential access files have a maximum of 255 available
      bytes per record and direct access files have a maximum of
      254 available bytes per record.
      A value of 0 implies a record length of 256.  If 'LENGTH=l'
      is omitted and the file type is sequential ('TYPE=S'),
      variable-length records are used.
 'KEY=k'  specifies the length of the key.  'k' is a numeric
      expression from 1 to 127.
 'KEY=k' must be used when the file type is ISAM ('TYPE=I'), and
      must be omitted for all other file types.
```

Note: MODE, TYPE, LENGTH, and KEY may appear in any order.

This statement sets up the required buffers and control blocks for disk file I/O. The file specified by 'file' is given a file-unit number. While the file is open, this number is used to reference the file.

A file cannot be opened under two file-units at once.

The parameters in the OPEN statement determine the file type, access mode, record length, and other specific features. See "Data Files" for a discussion of file access under RSBASIC.

Examples

```
    OPEN #1, "DATA/D", MODE=R, TYPE=D, LENGTH=32
```

Opens the file "DATA/D" for direct access, read-only, with a record length of 32. File-unit #1 will be used. If the file was created with a different record length, an error will occur

```
  OPEN #2, "MAILLIST/ISM", MODE=U, TYPE=I, LENGTH=128, KEY=25
```

Opens the file "MAILLIST/ISAM" for updating. The file must already exist on one of the diskettes in the system or an error will occur. The file must be indexed-sequential, with a record length of 128 and a key length of 25. File-unit #2 will be used.

```
    OPEN #( BASE% + CURNT% ), FILE$, MODE=E, TYPE=S
```

Opens the file specified by the contents of FILE$ for sequential writing beginning at the end of the file. The file-unit specified by the expression (BASE% + CURNT%) will be used.

Sample Program

See the chapter on data files.

-- FUNCTION --

OR
Calculate Logical OR

```
OR (number, number)
  'number' is any number in the range of
      [-32768, 32767].
```

OR is a logical operation performed on the binary representations of the two 'numbers'. OR searches the bits of each number to see if either or both are set to 1. A binary 1 is returned if either or both bits are 1; a 0 is returned only if neither bit contains a 1.

First Number	Second Number	Bit Returned
1	1	1
1	0	1
0	1	1
0	0	0

If 'number' is a real number, OR will convert it to an integer. The binary number returned is always expressed as an integer.

Note: See also AND and XOR.

Examples

```
   PRINT OR (192,3)
```

Prints 195. The operation is performed as follows:

```
                          Binary
        Integer       Representation
          192            11000000
            3            00000011
         ----           ---------
          195            11000011
```

```
    PRINT OR (195, 3)
```

Prints 195:

```
                                 Binary
      Integer                 Representation
       195                       11000011
         3                       00000011
       ---                       --------
       195                       11000011
```

Sample Program

```
 10 REM      *** SAMPLE PROGRAM DEMONSTRATING OR ***
 20 REM
 30 C$ = ""
 40 PRINT "TYPE A SENTENCE WITH UPPER AND LOWER CASE LETTERS"
 50 INPUT A$
 60 FOR X = 1 TO LEN(A$)
 70    B$ = SEG$(A$,X,1)
 80    D = ASC(B$)
 90    C$ = C$ & CHR$(OR(32,D))
100 NEXT X
110 PRINT "HERE IT IS IN ALL LOWER CASE : "; C$
120 GOTO 30
```

-- FUNCTION --

POS
Search for Specified String

```
POS(string 1, string 2)
  'string' is a string constant or a string variable:
     'string 1' is the string to be searched.
     'string 2' is the substring you want to search for.
```

Examples

In these examples, A$ = "LINCOLN"

   POS(A$, "INC")

Returns 2.

      POS(A$,"COLN")

Returns 4.

      POS(A$, "12")

Returns 0.

      POS(A$, "LINCOLNABRAHAM")

Returns 0.

Sample Program

```
 80 REM     ** SAMPLE PROGRAM DEMONSTRATING POS **
 90 REM
100 REM     ** SEARCH MAILING LIST FOR NO. OF *761 ZIP CODES **
110 REM
120 COUNTER = 0
130 READ ADDRESS$
140 IF ADDRESS$ = "0" THEN 170
```

```
   150 IF POS(ADDRESS$, "761") = 0 THEN 130
   160 COUNTER = COUNTER + 1
   170 PRINT "NUMBER OF TARRANT CO., TX ADDRESSES IS"; COUNTER : STOP
   180 DATA "5951 GORHAM DRIVE, BURLESON, TX *76148"
   190 DATA "71 FIRSTFIELD ROAD, GAITHERSGURG, MD *76102"
   200 DATA "16633 SOUTH CENTRAL EXPRESSWAY, RICHARDSON, TX *75080"
   210 DATA "1000 TWO TANDY CENTER, FORT WORTH, TX *76102"
   220 DATA "0"
 *RUN
 NUMBER OF TARRANT CO., TX ADDRESSES IS 1
```

-- STATEMENT --

PRINT
Print on Video Display

```
PRINT item-list
  'item-list' contains expressions to be evaluated and
      output to the video display. 'item-list' may
      also contain any of the special print functions
      listed below. Every item but the last must be
      followed by a semi-colon or comma.

      A semi-colon leaves the cursor in its current
      position; a comma advances the cursor to the next
      print zone (see description below).

      Unless a semi-colon or comma follows the last
      item, PRINT will output a carriage return
      after the last character is displayed.
```

This statement outputs to the display, beginning at the current cursor position. It outputs string data character-for-character, with no alteration, and modifies numeric data according to a default format described later on.

The punctuation between items (semi-colons or commas) determines the spacing between the text as it is displayed. A semi-colon produces no extra space, while a comma advances the cursor to the next print zone. The print zones are:

	ZONE 1	ZONE 2	ZONE 3	ZONE 4	ZONE 5
COLUMNS	0 15	16 31	32 47	48 63	64 79

Examples

```
    PRINT A / B
```

Displays the result of A/B.

```
    PRINT "THE SUM IS"; A + B
```

Displays the message in quotes followed by the result of A+B.

```
    PRINT "NAME", "AGE", "PHONE"
```

Displays the three headings in three successive print zones.

Cursor Motion and Print Positions

Whenever a character is printed in column 79, the cursor wraps around to column 0 on the next row. Whenever a character is printed in column 79 on the bottom row (23) of the display, the display scrolls up, and the cursor returns to column 0 of row 23. Scrolling also occurs when a carriage return or line-feed is printed while the cursor is anywhere on the bottom row.

(Scrolling: The text in row 1 is moved to row 0, the text in row 2 is moved to row 1, ... the text in row 23 is moved to row 22. The row 23 is then filled with blanks.)

The current cursor position determines where a particular item will be printed. In general, the current cursor position immediately follows the last character printed. However, there are several ways to move the cursor before printing an item.

Semi-Colons and Commas

When semi-colons are used as separators in the item list, each item is printed immediately after the last item printed. When commas are used as separators, the cursor advances to the next print zone after printing each item.

For example:

```
 10 DATA "FIRST", 100.100, "SECOND", 1234.567, "END", 0
 20 PRINT "DEMO OF PRINT WITH SEMI-COLONS IN ITEM-LIST"
 30 READ TXT$, NMBR
 40 PRINT TXT$; NMBR;
 50 IF TXT$ <> "END" THEN 30
 60 RESTORE
 70 PRINT: PRINT "DEMO OF PRINT WITH COMMAS IN ITEM-LIST
 80 READ TXT$, NMBR
 90 PRINT TXT$, NMBR,
100 IF TXT$ <> "END" THEN 80
```

Commas provide a convenient way of outputting tables to the display. The tables can contain up to five columns:

```
10 PRINT "N", "N^2", "N^3", "N^4", "N^5"
20 FOR N = 1 TO 5 STEP .5
30   PRINT N, N^2, N^3, N^4, N^5
40 NEXT N
```

CRT and CRTR

There are two special print functions for positioning the cursor. CRT moves it to an absolute row-column location; CRTR moves it to a relative row-column location, specified as an offset from the current row-column location. For syntax details, see CRT and CRTR.

Output Format for Numbers

. The value is rounded to a maximum of six significant digits (leading and trailing zeros are suppressed).
. After rounding, if the value is smaller than -999999 or greater than +999999, it is displayed in E-format, e.g.,
    1.1 E6 for the value 1100000
After rounding, if the value is greater than -0.0000001 and less than +0.0000001, it is displayed in E-format, e.g.,
    1.1 E-7 for the value 0.00000011
. Numbers between -1 and +1 which are not displayed in E-format are always displayed with a zero ahead of the decimal point, e.g.,
    0.05 for the value .05
. A single trailing space is always added to the number. A leading space is added if the number is positive and greater than zero.

Note: The PRINT USING statement lets you override these rules.

String Output

PRINT outputs in the scroll-mode. That means you can output any of the scroll-mode characters, including control characters. For a complete list of characters available, see the TRSDOS Reference Manual, Appendix 5, Column 4.

To send a character or string of characters, store the character(s) in a string variable and PRINT the variable. Or you can use the CHR$ and STRING$ functions. For example:

```
    A$ = "*****"
    PRINT A$
```

produces the same output as

```
    PRINT STRING$(5, "*")

    CLS$ = CHR$(27)
    PRINT CLS$
```

Stores control code 27 in CLS$. PRINTing CLSS clears the display and homes the cursor.

Graphics Characters

Since PRINT outputs in the scroll-mode, graphics characters cannot be output using a normal print list. Instead, there is a special function to provides graphics-mode ouptut. See CRTG. (For a list of graphics characters, see the TRSDOS Reference Manual, Appendix 5, column 5.

Other PRINT-related functions

TAB, CRTX, CRTY, CRTI.

-- STATEMENT --

PRINT to a disk file
Print to Disk

Sequential access:
```
  PRINT # file-unit; item-list
```
Indexed sequential:
```
  PRINT # file-unit, KEY=key; item-list
```
Direct access:
```
  PRINT # file-unit, KEY=record-number; item-list

    'file-unit'  is a numeric expression specifying
        the output file. The file-unit is assigned when
        the file is opened.
    'item-list'  contains expressions to be evaluated
        and output to the disk file. Every item but the
        last must be followed by a comma.  There
        should be no punctuation after the last item.
    'KEY=key'  is used for output to indexed sequential
        access files. 'key' is a string expression
        containing the sort key.
    'KEY=record-number'  is used for output to direct
        access files.  'record-number' is a numeric
        expression specifying the record number.
```

This statement performs disk output in a manner analogous to the PRINT to video display. Of course, none of the special video display functions may be used. One PRINT statement writes one record.

A comma ',' is inserted after each but the last item in the disk record.

For output formats, see PRINT to Video Display.

See "Data Files" for a discussion of file access under RSBASIC.

Examples

```
  PRINT #1; A+B
```

The value of A+B is output to file-unit #1.

```
  PRINT #2, KEY=NAME$; NAME$, PAYRAT, EXEMPT%
```

NAME$, PAYRAT, and EXEMPT are output to the record indexed by the the contents of NAME$, in file-unit #2.

```
  PRINT #3, KEY=RECNBR%; NAME$, PAYRAT, EXEMPT%
```

The same three items are output to record number RECNBR%, in file-unit #3.

Sample Program

See the chapter on data files.

-- STATEMENT --

PRINT USING
Print Using Format

```
PRINT USING image, KEY=record number; KEY=key;
    print-function, item list
  'image' specifies the format of the data; it can
      be a line number referring to an image
      statement, or a string expression containing
      the image.
  'print-function' is one of the special functions:
      CRT, CRTR or CRTG.  These functions position
      the cursor before printing starts. If omitted,
      printing starts at the current cursor position.
  'item-list' contains expressions to be evaluated
      and output to the video display.  A TAB function
      may be one of the items.  Every item but
      the last must be followed by a comma or semi-colon.
```

This statement outputs to the display, beginning at the current cursor location. Unlike PRINT, it outputs formatted data, according to an image specification contained on a separate program line or in a string expression.

When executed, PRINT USING attempts to output the first data item according to the first field in 'image', the second according to the second field, etc. If there are not enough image fields to satisfy the item-list, PRINT USING starts over at the beginning of 'image'.

Image Lines for PRINT USING

The image line indicates exactly how the data is to be printed: number of fields, length of each field, literal characters to insert between fields, and format for string or numeric fields. The following special characters are available for specifying the output format for string and numeric fields:

Special Character	Meaning
#	A numeric or string character. A sequence of N "#" characters represents a numeric or string field of N characters.
>	When used as the first character in a string field, data will be right-justified with truncation on the left.
<	When used as the first character in a string field, data will be left-justified with truncation on the right.
.	When used inside a numeric field, indicates the position of the decimal point.
,	When used inside a numeric field, specifies commas to be inserted at that position if a digit has already been printed.
-	When used ahead of a numeric field, a minus sign will be displayed ahead of negative numbers; blank space ahead of positive numbers.
+	When used ahead of a numeric field, a plus sign will be displayed ahead of positive numbers; minus sign ahead of negative numbers.
*	When used ahead of numeric fields, asterisks will be used as fill characters instead of the usual blanks.
$	When used ahead of numeric fields, the dollar sign will be displayed ahead of the number.
!!!!	When used following a numeric field, the number will be displayed with the same E notation that the Model II

BASIC Interpreter uses.

Any other characters--or any of the above characters used out of context--will be treated as literals and inserted into the display output. Such characters also serve as image-field delimiters (they mark the beginning and end of the fields).

If stored in a separate program line, image lines take this form:

```
line-number ;image
  line-number is a normal BASIC line number. (Image lines
      can be used anywhere in your program.
  ';' marks the line as a non-executable image line
  'image' is a sequence of characters defining the image
      format.
```

You can also store the image line inside a string, and then reference that variable in PRINT USING in place of the line-number.

Examples:

```
    100 IMAGE$ = "MR. ########## IS ## AND MAKES $#####.##"
    110 PRINT USING IMAGE$, NAME$, AGE%, SAL
```

Prints the values of the variables NAME$, AGE%, SAL using the image line stored in IMAGE$.

```
    100 ;MR. ########## IS ## AND MAKES $#####.##
    110 PRINT USING 100, NAME$, AGE%, SAL
```

Produces the same output as the previous example.

```
    110 PRINT USING 100, CRT(X%,Y%), NAME$, AGE%, SAL
```

Printing starts at row X%, column Y%.

```
    110 PRINT USING 100, NAME$, AGE%, SAL,
```

The trailing comma suppresses the usual carriage return after the last character is displayed.

How Data is Formatted into the Image

String Data

String data is left-justified into the image field, with filler blanks added on the right if necessary. If the string is too long to fit, the string is truncated on the right.

(When '>' is used as the first character in the field, the string is right-justified with filler blanks added on the left if necessary. If the string is too long to fit, truncation is on the left.)

Numeric Data

If the field contains a decimal point, the number is rounded to the precision specified in the image-field. The rounded numbers is always right justified, with filler blanks added on the left if necessary. If the number contains too many numeric characters to the left of the decimal point, a string of asterisks will be output to fill the field (no digits will be displayed.

Notes: Unless '+' or '-' is used ahead of the field, negative numbers will require one of the '#' positions for the sign. If '+' or '-' is used, the sign will not take one of the '#' positions.

If '*' is used, any unused leading positions will be filled with asterisks instead of with the usual blanks.

Sample Program

```
10 REM      *** PRINT USING ***
20 DIM IMAGE$80, STRNG$25
30 PRINT "ENTER THE OUTPUT IMAGE FOR THREE FIELDS: string, real, integer"
40 LINE INPUT IMAGE$
50 PRINT "NOW ENTER THE DATA: string, real-number, integer"
60 INPUT STRNG$, RLN, NTGR%
70 PRINT "HERE'S THE FORMATTED OUTPUT:"
80 PRINT USING IMAGE$, STRNG$; RLN; NTGR%
90 PRINT: GOTO 30
```

Sample Run

```
ENTER THE OUTPUT IMAGE FOR THREE FIELDS: string, real, integer
?############### #####.#### #####
NOW ENTER THE DATA: string, real-number, integer
?GEORGE, 1234.567 , 123
HERE'S THE FORMATTED OUTPUT:
GEORGE           1234.5670   123

ENTER THE OUTPUT IMAGE FOR THREE FIELDS: string, real, integer
?>############## #####.####
NOW ENTER THE DATA: string, real-number, integer
?GEORGE, 1234.567 , 1324
HERE'S THE FORMATTED OUTPUT:
         GEORGE  1234.5670
           1324

ENTER THE OUTPUT IMAGE FOR THREE FIELDS: string, real, integer
?>############### #####.#### ####
NOW ENTER THE DATA: string, real-number, integer
?GEROGE, 234.567 , 1234
HERE'S THE FORMATTED OUTPUT:
          GEROGE   234.5670 1234

ENTER THE OUTPUT IMAGE FOR THREE FIELDS: string, real, integer
?##########
NOW ENTER THE DATA: string, real-number, integer
?GEORGE, 1324.567 , 1234
HERE'S THE FORMATTED OUTPUT:
GEORGE
      1325
      1234
```

-- STATEMENT --

PRINT USING to a disk file
Print Using Format to Disk File

Sequential access:

```
  PRINT USING # file-unit; image, item-list
```

Indexed sequential:

```
  PRINT USING # file-unit, KEY=key; image, item-list
```

Direct access:

```
  PRINT USING # file-unit, KEY=record-number; image, item-list

   'file-unit'  is a numeric expression specifying
      the output file. The file-unit is assigned when
      the file is opened.
   'image' specifies the format of the data; it can be a
      line number referring to an image statement, or a
      string expression containing the image specifiers.
   'item-list'  contains expressions to be evaluated
      and output to the disk file. Every item but the
      last must be followed by a comma.  There
      should be no punctuation after the last item.
   'KEY=key'  is used for output to indexed sequential
      access files. 'key' is a string expression
      containing the sort key.
   'KEY=record-number'  is used for output to direct
      access files.  'record-number' is a numeric
      expression specifying the record number.
```

This statement performs disk output in a manner analogous to PRINT USING to video display. Of course, none of the special video display functions may be used.

PRINT USING outputs formatted data, according to an image specification contained on a separate line or in a string expression. When executed, it outputs the first data item according to the first field in 'image', the second, according to the second field, etc. If there are not enough image fields to satisfy the item-list, PRINT USING starts over at the beginning of 'image'.

For further details on image specifiers, see PRINT USING to Video Display. See "Data Files" for a discussion of file access under RSBASIC.

Examples

```
  PRINT USING #1; "###,###.##", A+B
```

The value of A+B is output using the specified format to file-unit #1.

```
  PRINT USING #2, KEY=NAME$; FMT$, NAME$; PAYRAT; EXEMPT%
```

NAME$, PAYRAT, and EXEMPT are output using the image in FMT$, to the record specified by the the contents of NAME$, to file-unit #2.

```
  100 ;<################### $##.## ##
  110 PRINT USING #3, KEY=RECNBR%; 100, NAME$; PAYRAT;
EXEMPT%
```

The same three items are output using the image of line 100, to record number RECNBR%, to file-unit #3.

Sample Program

See the chapter on data files.

-- STATEMENT --

RANDOMIZE
Reseed Random Number Generator

```
RANDOMIZE
```

RANDOMIZE reseeds the random number generator to a random place on the generator. If your program uses the RND function, the same sequence of pseudorandom numbers will be generated every time you Run the program. Therefore, you may want to put RANDOMIZE at the beginning of the program. This will help ensure that you get a different sequence of pseudorandom numbers each time you run the program.

RANDOMIZE needs to be executed only once in the program.

Example

```
RANDOMIZE
```

This statement helps ensure you will get a different sequence of random numbers every time you RUN the program.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING RANDOMIZE ***
 90 REM
100 RANDOMIZE
110 PRINT CHR$(27)
120 PRINT "PICK A NUMBER BETWEEN 1 AND 5"
130 INPUT A
140 B% = RND * 5 + 1
150 IF A = B THEN 180
160 PRINT "YOU LOSE, THE ANSWER IS "; B; " -- TRY AGAIN."
170 GOTO 120
180 PRINT "YOU PICKED THE RIGHT NUMBER -- YOU WIN!" : GOTO 120
```

```
PICK A NUMBER BETWEEN 1 AND 5
? 2
YOU LOSE, THE ANSWER IS  1  -- TRY AGAIN.
PICK A NUMBER BETWEEN 1 AND 5
? 1
YOU LOSE, THE ANSWER IS  2  -- TRY AGAIN.
PICK A NUMBER BETWEEN 1 AND 5
? 4
YOU LOSE, THE ANSWER IS  5  -- TRY AGAIN.
PICK A NUMBER BETWEEN 1 AND 5
? 3
YOU PICKED THE RIGHT NUMBER -- YOU WIN!
PICK A NUMBER BETWEEN 1 AND 5
?
```

-- STATEMENT --

READ
Get Value from DATA Statement

```
READ variable, ...
```

READ assigns a value from a DATA statement to the 'variable'. The first time READ is executed, READ assigns the first value in the first DATA statement to its first 'variable'. The second time, READ reads the second value in the first DATA statement and assigns it to its second variable. READ continues to assign data to its variables in sequential order moving to the second DATA statement when all the data in the first DATA statement has been read.

An Out of Data error occurs if there are more attempts to READ than there are DATA items.

NOTE: See also DATA.

Examples

```
READ T
```

Reads a numeric value from a DATA statement.

```
READ S$, T, U
```

Reads values for S$, T, and U from a DATA statement

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING READ ***
 90 REM
100 REM     *** READ IN DISCOUNT QUALIFICATIONS ***
110 READ Q1$, Q2$
```

```
  120 DATA "PRE-PAYMENT DISCOUNT", "QUANTITY DISCOUNT"
  130 REM     *** READ IN DISCOUNTS ***
  140 READ D1, D2
  150 DATA .05,.07
  160 REM
  170 PRINT Q1$; " --- "; D1*100; "%"
  180 PRINT Q2$; " --- "; D2*100; "%"
*RUN
PRE-PAYMENT DISCOUNT ---  5 %
QUANTITY DISCOUNT ---  7 %
```

-- STATEMENT --

READ from a disk file
Read Contents of Disk File

Sequential access files:
```
  READ # file-unit; variable-list
```

Indexed-sequential access files:
```
  READ # file-unit, KEY=key; variable-list
```

Direct access files:
```
  READ # file-unit, KEY=record-number; variable-list

    'file-unit'  is a numeric expression specifying
        the input file. The file-unit is assigned when
        the file is opened.
    'variable-list'  specifies the target variables to
        receive the data input from the file. Every
        variable but the last must be followed by a
        comma.  There should be no punctuation after
        the last variable.  If no variables are supplied,
        the current record is skipped.
    'KEY=key'  is used for input from indexed sequential
        access files. 'key' is a string expression
        containing the sort key.
    'KEY=record-number'  is used for input from direct
        access files.  'record-number' is a numeric
        expression specifying the record number.
```

This statement performs disk input of binary records written with the WRITE statement. 'variable-list' must match the 'item-list' used when the record was written, in number and type of data items. String variables must be large enough to contain string data; integer data must be read into integer variables; etc.

See "Data Files" for a discussion of file access under RSBASIC.

Examples

```
  READ  #1; A; B
```

Values for A and B are read from file-unit #1.

```
  READ #2, KEY=NAME$; PAYRAT, EXEMPT%
```

PAYRAT and EXEMPT are read from the record indexed by the the contents of NAME$, in file-unit #2.

```
  READ #3, KEY=RECNBR%; PAYRAT, EXEMPT%
```

The same two items are read from record number RECNBR%, in file-unit #3.

Sample Program

See the chapter on data files.

-- STATEMENT --

REAL
Define Variables as Real Numbers

```
REAL*8 letter-list
   *8 represents the eight byte length of real
      numbers.  This may be omitted.
   'letter-list' is a sequence of individual
      letters or letter-ranges; the elements
      in the list must be separated by commas.
      A letter-range is in the form:
         'letter1-letter2'
```

REAL defines all variables, or all beginning with the letters specified in 'letter-list' as real. However, a type declaration character will override the REAL statement. Real numbers are stored in 8-bytes and have 14 digits of precision, although only 6 are printed.

REAL with a letter list may be used after an INTEGER or STRING statement to override the integer or string defaults for certain specified variable names. For example:

```
10 INTEGER
20 REAL A-C
```

causes all variables, except those beginning with the letters A through C to be integers. Variables beginning with A, B, and C are real.

Note: For more information, see the chapter on BASIC Concepts.

Examples

```
REAL I, W-Z
```

Causes any variables beginning with the letters I or W through Z to be real variables. However, I% would still be an integer variable because of its type declaration tag.

Sample Program

```
100 REM    *** REAL STATEMENT ***
110 INTEGER
120 REAL X-Z    :REM   NOW X, Y AND Z ARE IMPLICIT REAL; OTHERS ARE INTEGER
130 INPUT PROMPT="TYPE IN A REAL NUMBER: "; X
140 A = X
150 PRINT "REAL NUMBER IS "; X
160 PRINT "CONVERTED TO INTEGER IS "; A
```

-- STATEMENT --

REM
Comment Line (Remarks)

```
   REM
```

REM instructs the Computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you or someone else looks at a listing of your program, it will be easier to figure out.

The apostrophe (') may be substituted for REM.

Examples

```
   REM  This is a remark
   REM
   REM  **********************
   '    This is a remark
```

All of these lines will be ignored when the program is executed.

```
   X=1      :   REM Initialize X
   X=X+1    :   REM Increment X
```

Both statements on the right side of the colon will be ignored when the program is executed.

Sample Program

```
10 REM      *** SAMPLE PROGRAM DEMONSTRATING REM ***
20 REM
30 INPUT A                              : REM  Input real number
40 A = A/2                              : REM  Find smaller values
50 IF A < 1E-8 THEN PRINT "That's small enough!": STOP
60 PRINT A                              : REM  Print
70 GOTO 40                             : REM  Loop for next
```

-- STATEMENT --

RESET BREAK
Disable the <BREAK> Handling Routine

```
   RESET BREAK
```

RESET BREAK disables the <BREAK> handling routine you set up with ON BREAK GOTO.

For example, you might use ON BREAK GOTO so that a person's pressing the <BREAK> key will be handled a certain way at the first of your program. However, in the second part of your program you might want BASIC to handle <BREAK> in the normal way. You may then use RESET BREAK to get BASIC to ignore the ON BREAK GOTO statement.

Note: See also ON BREAK GOTO

Example

```
   RESET BREAK
```

Causes BASIC to ignore the previous ON BREAK GOTO statement and handle <BREAK> in the normal way.

Sample Program

See ON BREAK GOTO.

-- STATEMENT --

RESET ERROR
Disable Error Handling

```
RESET ERROR
```

RESET ERROR disables an ON ERROR GOTO statement. Although ON ERROR GOTO is disabled every time it is used, RESET ERROR disables an ON ERROR GOTO statement that has not yet been used.

NOTE: See also ON ERROR GOTO, ERR, ERROR, and RESET GOSUB.

Example

If you are using ON ERROR GOTO to trap a possible error in one part of the program, but don't want any errors trapped in another part of the program:

```
RESET ERROR
```

Would cause the ON ERROR GOTO statement to be ignored.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING RESET ERROR ***
 90 REM
100 ON ERROR GOTO 180
110 PRINT "INPUT A NUMBER"
```

```
  120 INPUT A
  130 RESET ERROR
  140 PRINT "THE NEXT ERROR IN THIS PROGRAM"
  150 PRINT "WILL BE HANDLED IN THE NORMAL WAY"
  160 PRINT A/0
  170 STOP
  180 IF ERR <> 5 THEN ERROR ERR
  190 PRINT "YOU MAY ONLY INPUT A NUMBER"
  200 ON ERROR GOTO 180
  210 GOTO 110
*RUN
INPUT A NUMBER
? 789
THE NEXT ERROR IN THIS PROGRAM
WILL BE HANDLED IN THE NORMAL WAY

DIVISION BY ZERO ERROR LINE 160
 1. E+63
```

-- STATEMENT --

RESET GOSUB
Clear All Returns

```
   RESET GOSUB
```

Whenever GOSUB is used, the Computer must store the return address. Normally, this return address is cleared when the RETURN statement is executed.

However, if an error handling routine is executed, these return addresses might never be cleared. By using the RESET GOSUB statement in your error handling routine, BASIC will clear all of these return addresses.

Note: See also ON ERROR GOTO, GOSUB, and RETURN.

Examples

```
   RESET GOSUB
```

This statement clears all return addresses.

Sample Program

```
10 REM    *** RESET GOSUB STATEMENT ***
15 DIM S$1
20 ON ERROR GOTO 1000
30 PRINT "SELECT OPTION 1, 2, OR 3: ";
40 S$ = INPUT$(2)
```

```
   50 O% = VAL%(S$)
   60 ON O% GOSUB 100, 200, 300
   70 GOTO 30
  100 PRINT "OPTION 1"
  110 RETURN
  200 PRINT "OPTION 2"
  210 RETURN
  300 PRINT "OPTION 3"
  310 RETURN
 1000 RESET GOSUB
 1010 GOTO 30
*RUN
SELECT OPTION 1, 2, OR 3: 2
OPTION 2
SELECT OPTION 1, 2, OR 3: 3
OPTION 3
SELECT OPTION 1, 2, OR 3:
```

-- STATEMENT --

RESTORE
Reset Data Pointer

```
RESTORE line number
```

When the Computer is READing data, it will read the data from the DATA statements sequentially and quit reading when all the data has been read. This means that without RESTORE, you can only use each data item once.

RESTORE causes the next READ statement to start over in reading the first item in the first DATA statement again. If you specify a line number it will start over reading the first data item on that particular DATA line.

Examples

```
RESTORE 300
```

The next READ statement will begin reading the first data item on the DATA statement at line 300.

```
RESTORE
```

The next READ statement will begin reading the first data item on the first DATA statement line.

Sample Program

```
80 REM      *** SAMPLE PROGRAM DEMONSTRATING RESTORE ***
90 REM
95 REM
```

```
100 REM     *** READ IN PROMPTS ***
105 REM
110 DATA "TRY ANOTHER ANSWER","KEEP TRYING","IT BEGINS WITH AN A","LAST
120 READ PROMPT$
130 IF PROMPT$ = "LAST" THEN RESTORE: GOTO 120
140 REM
145 REM
150 REM     *** BEGIN GEOGRAPHY EDUCATION PROGRAM ***
155 REM
160 PRINT "WHAT IS THE CAPITAL OF TEXAS"
170 INPUT A$
180 IF A$ <> "AUSTIN" THEN PRINT PROMPT$: GOTO 120
190 PRINT "VERY GOOD..THAT'S THE ONLY QUESTION WE HAVE FOR NOW..."
```

-- STATEMENT --

RESUME
Terminate Error-Trapping Routine

```
RESUME
  Exection resumes at the beginning of the statement
    causing the error.

RESUME NEXT
  Execution resumes after the statement causing the
    error.
```

RESUME terminates an error-handling routine by specifying where normal execution is to resume. Place a RESUME statement at the end of an error-trapping routine. That way later errors can also be trapped.

RESUME causes the Computer to return to the statement in which the error occured. RESUME NEXT causes the Computer to branch to the statement following the point at which the error occurred.

Example

```
   RESUME
```

If an error occurs, when program execution reaches the line above, control will be transferred to the statement in which the error occurred.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING RESUME ***
 90 REM
100 ON ERROR GOTO 500
110 READ A
```

```
120 PRINT A,
130 GOTO 110
140 DATA 1, 2, 3, 4, 5, 6
150 STOP
500 IF ERR <> 7 THEN ERROR ERR
510 PRINT "DO YOU WANT TO PRINT THE LIST AGAIN"
520 INPUT R$
530 IF R$ = "NO" THEN STOP
540 RESTORE
550 ON ERROR GOTO 500
560 RESUME
```

-- STATEMENT --

RETURN
Return Control to Calling Program

```
RETURN
```

RETURN ends a subroutine by returning control to the statement immediately following the most-recently executed GOSUB. If RETURN is encountered without execution of a matching GOSUB, an error will occur.

Example

```
RETURN
```

This line ends the subroutine, returning execution back to the line immediately following the most recently executed GOSUB.

Sample Program

```
10 REM     *** SAMPLE PROGRAM DEMONSTRATING RETURN ***
20 REM
30 PRINT "THIS PROGRAM FINDS THE AREA OF A CIRCLE"
40 PRINT "TYPE IN A VALUE FOR THE RADIUS"
50 INPUT R
60 GOSUB 80
70 PRINT "AREA IS"; A: STOP
80 A = 3.14 * R * R
90 RETURN
```

-- FUNCTION --

RND
Generate Pseudorandom Number

```
RND
RND(number)
   'number' is a positive integer.
```

RND produces a pseudorandom number between 0 and 1. Programmers commonly use it to introduce the element of chance in a program.

This random number is generated by using the current "seed"number. When you specify a 'number' with RND, RND reseeds the generator with that 'number'. To reseed the generator at random, use the RANDOMIZE statement.

RND always returns a real number between 0 and 1. The examples below show how to produce random integers higher than 1.

Examples

```
   PRINT RND
```
Prints a random number between 0 and 1.

```
   PRINT RND * 2
```

Prints a random number between 0 and 2.

```
   PRINT INT(RND * 2)
```

Prints either 0 or 1 at random.

```
   PRINT INT(RND * 2 + 1)
```

Prints either 1 or 2 at random.

```
   PRINT INT(RND * 100 + 1)
```

Prints a random whole number between 1 and 100.

```
A = RND
```

A random number between 0 and 1 is assigned to A.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING RND ***
 90 REM
100 PRINT: PRINT: PRINT "ROLLING THE DICE .........."
110 X = INT(RND(0) * 5) + 1
120 Y = INT(RND(0) * 5) + 1
130 PRINT: PRINT "YOUR ROLL IS"; X; "AND"; Y; "----------"; X+Y
140 FOR A = 1 TO 450: NEXT A
150 PRINT: GOTO 100
```

-- FUNCTION --

SEG$
Get Substring

```
SEG$(string, position, length)
    'string' is a string constant or a string variable.
    'position' is the position where the substring
       begins in the 'string'.
    'length' is the number of characters in the
       substring.  If omitted, the length from
       position to the end of 'string' is used.
```

SEG$ returns a substring of 'string'. The substring begins at 'position' in the 'string' and is 'length' characters long.

Examples

If A$ = "WEATHERFORD" then

```
   PRINT SEG$(A$, 3, 2)
```

Prints AT

```
   F$ = SEG$(A$, 3)
```

Puts ATHERFORD into F$.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING SEG$ ***
 90 REM
100 PRINT "AREA CODE AND NUMBER (NNN-NNN-NNNN)"
110 INPUT PH$
120 EX$ = SEG$(PH$,5,3)
130 PRINT "NUMBER IS IN THE "; EX$; " EXCHANGE"
140 GOTO 100
```

-- FUNCTION --

SGN
Get Sign

```
   SGN(number)
      'number' is a numeric expression
```

This function returns the sign of the 'number'. It returns a 1 if the number is positive, 0 if it is a 0, and -1 if it is negative.

Examples

```
   PRINT SGN(5)
```

Prints 1.

```
   PRINT SGN(-5)
```

Prints -1.

```
   PRINT SGN(0)
```

Prints 0.

```
   Y = SGN(A * B)
```

Determines the value of A * B and assigns the appropriate number (-1, 0, 1) to Y

```
   PRINT SGN(N)
```

Prints the appropriate number.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING SGN ***
 90 REM
100 PRINT "ENTER A NUMBER"
110 INPUT X
120 ON SGN(X) + 2 GOTO 130, 140, 150
130 PRINT "NEGATIVE" : STOP
140 PRINT "ZERO" : STOP
150 PRINT "POSITIVE" : END
```

-- FUNCTION --

SIN
Compute Sine

```
   SIN(number)
     'number' is a numeric expression.
```

SIN returns the sine of the 'number', which must be in radians. To obtain the sine of X when X is in degrees, use SIN(X * .01745329251993).

The result is always a real number.

Examples

```
   W = SIN(MX)
```

Assigns the value of SIN(MX) to W.

```
   PRINT SIN(7.96)
```

Prints the value .994385.

```
   E = (A * A) * (SIN(D)/2)
```

Performs the indicated calculation and stores it in E.

NOTE: Trigonometric functions are not loaded when you load the BASIC Compiler; they are loaded upon demand. This might cause a slight delay when using these functions, since they must be loaded into the system first.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING SIN ***
 90 REM
100 PRINT "INPUT ANGLE IN DEGREES"
110 INPUT A
120 PRINT "SINE IS"; SIN(A * .01745329)
130 GOTO 100
```

-- FUNCTION --

SQR
Compute Square Root

```
   SQR(number)
     'number' is a non-negative numeric expression.
```

SQR returns the square root of the 'number'. The result is always a real number.

If 'number is a negative value, SQR will print a warning and then return the square root of the absolute value of 'number'.

Examples

```
   PRINT SQR(9)
```

Prints 3.

```
   PRINT SQR(6 + 3)
```

Prints 3.

```
   PRINT SQR(155.7)
```

Prints 12.478.

```
   Y = SQR(A * B)
```

Assigns the value of the square root of A * B to Y.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING SQR ***
 90 REM
100 PRINT "NUMBER", "SQUARE ROOT", "", "NUMBER", "SQUARE ROOT"
110 FOR X = 1 TO 44 STEP 2
120    PRINT X, SQR(X), "", X + 1, SQR(X + 1)
130 NEXT X
140 GOTO 140
```

-- STATEMENT --

STOP
Stop Program Execution

```
STOP
```

STOP terminates execution of your program at the line number you specify. Normally, STOP is used to terminate execution at a line other than the end of the program.

Unlike END, the compiler will compile the entire program including the lines following the STOP statement. However, when the program is executed, no lines after STOP will be executed.

NOTE: STOP is used in the same manner END is used with the BASIC Interpreter.

Example

```
   STOP
```

This line is the last line executed. No lines following it are executed.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING STOP ***
 90 REM
100 PRINT "DO YOU WANT TO CONTINUE"
110 INPUT A$
120 IF A$ = "YES" THEN 140
130 STOP
140 PRINT "THE REST OF THE PROGRAM"
```

-- FUNCTION --

STR$
Convert to String Representation

```
STR$(number, image)
   'number' is a numeric expression.
   'image' specifies the format of 'number'.  It
      can be a string variable containing the
      image or a string constant.  If omitted,
      'number' is printed as a real number with
      6 digits of precision.
```

STR$, the inverse of VAL, converts the 'number' to a string. For example, if X = 58.5, then STR$(X) equals the string " 58.5". Notice that a leading blank is inserted before 58.5 to allow for its sign.

While numeric operations (such as addition, subtraction, multiplication and division) may be performed on X, only string functions and operations may be performed on the string, " 58.5".

You may use an image with STR$ to specify the format in which you want the number printed. See PRINT USING for information on how to construct an image. If you don't use an image, the number will be printed in the real number format. See PRINT for an explanation on how real numbers are printed.

Examples

```
   A$ = STR$(100) & " DOLLARS"
```

Assigns "100 DOLLARS" to A$>

```
   PRINT "NUMBER " & STR$(6+3)
```

Prints NUMBER 9.

```
   S$ = STR$(X)
```

Converts the number X into a string and stores it in S$.

```
   PRINT STR$(10000000)
```

Prints 1.E+7. (See PRINT for an explantion of the E notation).

```
   A$ = STR$(35592163)
```

Assigns "35592163" to A$.

```
   PRINT STR$(600000000, "#########")
```

Prints "600000000"

```
   PRINT STR$(60000000)
```

Prints 6.E+8

```
   PRINT STR$(35.24, A$)
```

Prints "35.24" in the format contained in A$.

Sample Programs

```
    5 REM      *** SAMPLE PROGRAM DEMONSTRATING STR$ ***
    6 REM
   10 PRINT "INPUT ITEM NUMBER"
   15 INPUT ITEM
   20 PRINT "INPUT COST OF ITEM"
   25 INPUT COST
   30 PRICE = COST * 2.5
   40 CODE$ = "I" & STR$(ITEM) & "C" & STR$(COST) & "P" & STR$(PRICE)
   50 PRINT "ITEM IS NOW CODED AS "; CODE$ : STOP
*RUN
INPUT ITEM NUMBER
? 2
INPUT COST OF ITEM
? 3.00
ITEM IS NOW CODED AS I2C3P7.5
```

```
   10 PRINT "TYPE A NUMBER WITH 14 DIGITS OR LESS"
   20 INPUT A
   30 PRINT "THE NUMBER WITHOUT THE FORMAT IS PRINTED :"; STR$(A)
   40 PRINT "THE NUMBER WITH THE FORMAT '#######.#######' IS :";
   50 PRINT STR$(A,"#######.#######")
*RUN
TYPE A NUMBER WITH 14 DIGITS OR LESS
? 3333333.3333333
THE NUMBER WITHOUT THE FORMAT IS PRINTED :3.33333E+6
THE NUMBER WITH THE FORMAT '#######.#######' IS :3333333.3333333
```

-- STATEMENT --

STRING
Define Variables as Strings

```
STRING*length letter-list
    *'length' is the number of characters which will be
      alloted for each string variable.
      If omitted, all string variables will
      be stored as 255 characters (255 bytes).
    'letter list' is a sequence of individual letters
      or letter-ranges; the elements in the list must
      be separated by commas.  A letter-range is in the
      form:
         letter1 - letter2
```

STRING causes all variables in the program to be classified as string unless a type declaration tag is used. All string variables will be stored as if they have 255 characters unless you specify a length.

If you use 'letter-list', only variable names beginning with those letters will be classified as string.

Note: For more information, see the chapter on BASIC Concepts.

Example

```
   STRING C, L-Z
```

Causes any variables beginning with the letters C or L through Z to be string variables, unless a type declaration is added. Each of these variables will be stored as a 255-character string.

```
   STRING
```

Causes all variables to be 255-character string variables, unless a type declaration tag is used.

```
   STRING*5
```

Causes all variables to be 5-character string variables, unless a type declaration tag is used.

```
   STRING*1 A-F
```

Causes all variables beginning with the letters A through F to be 1-character string variables unless a type declaration tag is used.

Sample Program

```
    10 REM    *** STRING STATEMENT ***
    20 STRING*80 L
    30 STRING*1  C
    40 PRINT "TYPE IN A MESSAGE"
    50 INPUT L
    60 PRINT "TYPE IN A SINGLE CHARACTER ";
    70 C = INPUT$(1)
    80 PRINT "THE MESSAGE WAS: "; L
    90 PRINT "THE CHARACTER WAS: "; C
 *RUN
 TYPE IN A MESSAGE
 ? THIS IS A MESSAGE
 TYPE IN A SINGLE CHARACTER K
 THE MESSAGE WAS: THIS IS A MESSAGE
 THE CHARACTER WAS: K
```

-- FUNCTION --

STRING$
Return String of Characters

```
STRING$(length, character)
   'length' is numeric expression in the range of
      0 to 255.
   'character' is a string constant or a string
      variable.
```

STRING$ is useful for creating graphs or tables, where you want to print a large string of the same characters. It returns a string of the character you specify. How many characters are in the string depends on the length you specify.

Examples

```
   PRINT STRING$(10, "-")
```

Prints ----------.

```
   B$ = STRING$(25, "X")
```

A string of 25 X's - XXXXXXXXXXXXXXXXXXXXXXXXX - is stored as B$.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING STRING$ ***
 90 REM
100 PRINT CHR$(27) : X = -2
110 PRINT CRT(0, 30); "SALES OF EACH ITEM"
120 FOR I = 1 TO 6
130      READ A : X = X + 4
140      PRINT CRT(X,0); "ITEM "; I; "  "; STRING$(A, "X")
150 NEXT I
160 GOTO 160
170 DATA 35,44,70,24,62,13
```

-- STATEMENT --

SUB
Name and Define Subprogram

```
SUB "subname"; dummy variable list
  'subname' is a 1 to 6 character string constant
  'dummy variable list' consists of any kind of
     variables separated by commas.
```

SUB must always be the first statement in a subprogram. It names the subprogram and lists its dummy variables. These dummy variables are given the values of whatever variables or constants are passed from the main program in the CALL statement.

For instance, if the SUB statement lists the dummy variable X (SUB "SUB"; X), and the CALL statement sends it the value Y (CALL "SUB"; Y), X will be given the value Y.

The type of dummy variables in the SUB statement must match the type of variables in the corresponding CALL statement.

Note: No other statement can precede SUB except for END or SUBEND. (If you want to insert remark lines, insert them after SUB.)

Examples

```
SUB "DEPREC"; A, B
```

This is the first line of the subprogram named "DEPREC". The dummy variables are A and B. They will be contain the value of whatever variables, expressions, or constants are sent to them by the CALL statement in the main program.

```
SUB "TABLE"; A$, B$, C, D, E( , )
```

Initiates and defines the subprogram named "TABLE". The dummy variables are A$, B$, C, D, E(,).

```
SUB "GRAPH"; HORZ, VERT
```

Initiates and defines the subprogram named "GRAPH". The dummy variables are HORZ and VERT

NOTE: For more information on subprograms see the Section on Segmenting Programs. See also CALL, END, and SUBEND.

Sample Program

```
   80 REM     *** SAMPLE PROGRAM DEMONSTRATING SUB ***
   90 REM
  100 A$ = "312/327-0092"
  110 B$ = "214/555-1313"
  120 PRINT "TELEPHONE NUMBERS :"
  130 PRINT A$: PRINT B$
  140 CALL "AREA"; A$
  150 CALL "AREA"; B$
  160 PRINT "THE AREA CODES ARE "; A$; " AND "; B$
  170 END
  180 SUB "AREA"; T$
  190 T$ = SEG$(T$,1,3)
  200 SUBEND
*RUN
TELEPHONE NUMBERS :
312/327-0092
214/555-1313
THE AREA CODES ARE 312 AND 214
```

-- STATEMENT --

SUBEND
End Subprogram

```
SUBEND
```

SUBEND is the last statement in the subprogram. It returns execution back to the statement in the main program immediately following the statement which CALLed the subprogram.

Example

```
   SUBEND
```

returns control back to the main program.

There must be only one SUBEND per subprogram.

NOTE: For more information on subprograms, see the section on Subprograms. See also CALL, END, and SUB.

Sample Program

```
  80 REM      *** SAMPLE PROGRAM DEMONSTRATING SUBEND ***
  90 REM
 100 X = RND(Ø)
 110 Y = RND(Ø)
 115 PRINT "BEFORE EXECUTING THE SUBROUTINE"
 117 PRINT "X ="; X; " AND Y ="; Y
 120 CALL "RAND"; X
 130 CALL "RAND"; Y
 140 PRINT "AFTER EXECUTING THE SUBROUTINE"
 142 PRINT "X="; X; " AND Y ="; Y
 160 END
 170 SUB "RAND"; A
 180 A = CVI(A * 100)
 190 SUBEND
*RUN
```

```
BEFORE EXECUTING THE SUBROUTINE
X = 0.922525  AND Y = 0.269398
AFTER EXECUTING THE SUBROUTINE
X= 92  AND Y = 26
```

-- STATEMENT --

SWAP
Exchange Values of Variables

```
   SWAP variable1, variable2
```

The SWAP statement allows the values of two variables to be exchanged. Either or both of the variables may be elements of arrays. Both variables must be the same type or a Type Mismatch error will result.

Example

```
   SWAP F1, F2
```

The contents of F2 are put into F1, and the contents of F1 are put into F2.

Sample Program

```
10 REM      *** SAMPLE PROGRAM DEMONSTRATING SWAP ***
20 REM
30 REM      *** BUBBLE SORT USING SWAP ***
40 REM
50 INTEGER A-Z: DIM A(50)
60 A(0) = 0
70 PRINT "HERE ARE 50 NUMBERS BETWEEN 1 AND 100"
80 FOR I = 1 TO 50: A(I) = CVI(RND(0)*100+1): PRINT A(I); : NEXT
90 PRINT: PRINT: PRINT "NOW SORTING DATA. START TIME = "; TAB(40); TIME$
```

```
  100 F = 0: K = 0          : REM F is set when a SWAP is make, K is counter
  109 REM      *** swap and set F ***
  110 IF A(K) > A(K+1) THEN SWAP A(K), A(K+1): F = 1
  120 K = K + 1: IF K < 50 THEN 110
  129 REM      *** go through data again until F = 0 ***
  130 IF F = 1 THEN 100
  140 PRINT: PRINT "DATA SORTED.  END TIME = "; TAB(40); TIME$
  150 PRINT: PRINT "HERE IT IS IN ORDER: "
  160 FOR I = 1 TO 50: PRINT A(I); : NEXT
*RUN
HERE ARE 50 NUMBERS BETWEEN 1 AND 100
 93  27  57  6  49  52  36  54  46  24  84  61  77  66  71  81  93  44  14  14
77  68  4  13  23  83  51  68  98  93  61  11  18  17  93  83  8  91  25  11  92
  67  20  4  5  88  90  37  28  55

NOW SORTING DATA. START TIME =
                                           06.00.48

DATA SORTED.  END TIME =
                                           06.00.54

HERE IT IS IN ORDER:
 4  4  5  6  8  11  11  13  14  14  17  18  20  23  24  25  27  28  36  37  44
 46  49  51  52  54  55  57  61  61  66  67  68  68  71  77  77  81  83  83  84
88  90  91  92  93  93  93  93  98
STOP LINE 160
```

-- STATEMENT --

SYSTEM
Execute a TRSDOS Command

```
SYSTEM "command"
    'command' is a TRSDOS Library command.  It may be
       a string constant of a string variable.  If 'command'
       is omitted, the system returns to TRSDOS.
```

There are times during your BASIC program where you might want the Computer to execute a TRSDOS command. For instance, you might want the TRSDOS "DIR" command to display a directory of the diskette in the middle of a BASIC program. To do this, you may use SYSTEM.

After executing the TRSDOS command, the Computer will return back to BASIC and execute the next statement in your program.

See the Model II Disk Operating System manual for a list of all the TRSDOS commands.

Example

```
   SYSTEM "DIR"
```

The Computer displays the directory and returns back to the next BASIC statement.

Sample Program

```
10 REM    *** SYSTEM COMMAND ***
20 DIM C$80
30 PRINT "HERE ARE THE SYSTEM LIBRARY COMMANDS AVAILABLE:"
40 SYSTEM "LIB"
50 LINE INPUT PROMPT="TYPE IN ONE OF THEM: "; C$
60 SYSTEM C$
```

-- FUNCTION --

TAB
Tab to Position

```
TAB(number)
    'number' is a numeric expression in the range
    of [0,255].  If greater than 255 BASIC reduces
    it MOD 255. IIf 'number' is less than the
    current cursor position, BASIC goes to the next
    line and TABS to position.
```

TAB, used in a PRINT statement moves the cursor to the column position specified. TAB may only be used in a PRINT statement.

Note: See CRT for an illustration of the 80 column positions on the video display.

Examples

```
PRINT TAB(5);"TABBED 5";
```

This prints:

```
 TABBED 5
```

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING TAB ***
 90 REM
100 PRINT CHR$(27)
110 PRINT TAB(2); "CATALOG NO."; TAB(16); "DESCRIPTION OF ITEM";
120 PRINT TAB(39); "QUANTITY"; TAB(51); "PRICE PER ITEM";
130 PRINT TAB(69); "TOTAL PRICE"
```

-- FUNCTION --

TAN
Compute Tangent

```
TAN(number)
  'number' is a numeric expression.
```

TAN returns the tangent of the 'number'. The number must be in radians. To obtain the tangent of X when X is in degrees, use TAN(X * .01745329251994). The result is always a real number.

Examples

```
L = TAN(M)
```

Assigns the value of TAN(M) to L.

```
PRINT TAN(7.96)
```

Prints the value -9.39696.

```
Z = (TAN(L2 - L1))/2
```

Performs the indicated calculation and stores the result in Z.

NOTE: Trigonometric functions are not loaded when you load the BASIC Compiler; they are loaded upon demand. This might cause a slight delay when using these functions, since they must be loaded into the system first.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING TAN ***
 90 REM
100 PRINT "INPUT ANGLE IN DEGREES"
110 INPUT ANGLE
120 T = TAN(ANGLE * .01745329)
130 PRINT "TANGENT IS"; T
140 GOTO 100
```

-- FUNCTION --

TIME$
Get the Time

```
TIME$
```

This function lets you use the time in a program.

The operator sets the time initially when TRSDOS is started up. When you request the time (with PRINT TIME$), BASIC will supply it using this format:

```
   14.47.18
```

which means 14 hours, 47 minutes, and 18 seconds (24-hour clock) or 2:47:18 PM.

To change the time, use the TRSDOS command, TIME. For example:

```
   SYSTEM "TIME 13.30"
```

sets the time to 13 hours and 30 minutes (and 0 seconds) or 1:30 PM.

Even if the operator never sets the time, TRSDOS will record the time at 00.00.00 when the system is started up and keep a record of how much time has passed.

Examples

```
   PRINT TIME$
```

Prints the time.

```
   A$ = TIME$
```

When this line is reached in your program, the current time is stored as A$.

Sample Program

```
 80 REM      *** SAMPLE PROGRAM DEMONSTRATING TIME$ ***
 90 REM
100 T$ = TIME$ : IF SEG$(T$,1,5) = "10.15" THEN 120
110 GOTO 100
120 PRINT "TIME IS 10:15 A.M. -- TIME TO PICK UP THE MAIL"
```

-- FUNCTION --

VAL
Evaluate String

```
VAL(string)
   'string' is a string constant or a string variable.
```

VAL is the inverse of STR$. It converts the characters in the 'string' to their numeric value. VAL returns a real number. VAL% returns an integer.

VAL quits looking for numeric characters as soon as it hits a character that has no meaning. For instance VAL(10Z5) returns a 10 -- it stopped its search when it encountered the Z and returned 10, the current numeric value.

If the string contains no numbers or is null (has a length of zero), VAL returns a 0.

Examples

```
   PRINT VAL("100 DOLLARS")
```

Prints 100.

```
   PRINT VAL("100 DOLLARS AND 50 CENTS")
```

Prints 100.

```
   PRINT VAL("1234E8")
```

Prints 1234E+8 (1234 * 10 ^ 8)

```
   PRINT VAL("ONE")
```

Prints 0.

```
   X = VAL("12.58")
```

Assigns the number, 12.58 to X.

```
   A = VAL(B$)
```

Assigns the numeric value of B$ to A.

```
   PRINT VAL%("12.58")
```

Prints 12

Sample Program

```
     80 REM      *** SAMPLE PROGRAM DEMONSTRATING VAL ***
     90 REM
    100 REM      *** WHAT SIDE OF THE STREET? ***
    110 REM      *** NORTH IS EVEN; SOUTH IS ODD ***
    120 REM
    125 PRINT "ENTER THE ADDRESS (NUMBER AND STREET) "
    130 LINE INPUT AD$
    140 C = CVI(VAL(AD$)/2) * 2
    145 PRINT C, VAL(AD$)
    150 IF C = VAL(AD$) THEN PRINT "NORTH SIDE" : GOTO 125
    160 PRINT "SOUTH SIDE" : GOTO 130
  *RUN
  ENTER THE ADDRESS (NUMBER AND STREET)
  ? 3200 ASH PARK
   3200             3200
  NORTH SIDE
  ENTER THE ADDRESS (NUMBER AND STREET)
  ? 3205 ASH PARK
   3204             3205
  SOUTH SIDE
  ?
```

-- STATEMENT --

WRITE to a disk file
Write to Disk

Sequential access files:

```
  WRITE # file-unit; item-list
```

Indexed-sequential access files:

```
  WRITE # file-unit, KEY=key; item-list
```

Direct access files:

```
  WRITE # file-unit, KEY=record-number; item-list

    'file-unit'  is a numeric expression specifying
        the output file. The file-unit is assigned when
        the file is opened.
    'item-list'  contains expressions to be evaluated
        and output to the disk file. Every item but
        the last must be followed by a comma.
        There should be no punctuation after the last
        item.  If item list is empty, the record is
        written as a deleted record.
    'KEY=key'  is used for output to indexed sequential
        access files. 'key' is a string expression
        containing the sort key.
    'KEY=record-number'  is used for output to direct
        access files.  'record-number' is a numeric
        expression specifying the record number.
```

This statement performs disk output of binary records for subsequent input by an analogous READ statement. 'item-list' must match the 'item-list' to be used when the record is read, in number and type of data items.

See "Data Files" for a discussion of file access under RSBASIC.

Examples

```
  WRITE  #1; A+B
```

The

value of A+B is written to file-unit #1.

```
  WRITE #2, KEY=NAME$; PAYRAT, EXEMPT%
```

PAYRAT and EXEMPT are written to the record indexed by the contents of NAME$, in file-unit #2.

```
  WRITE #3, KEY=RECNBR%; PAYRAT, EXEMPT%
```

The same two items are written to record number RECNBR%, in file-unit #3.

Sample Program

See the chapter on data files.

-- FUNCTION --

XOR
Calculate Exclusive OR

```
XOR(number, number)
  'number' is any integer in the range of -32768 to
     32767.
```

XOR is a logical operation performed on the binary representations of the two 'numbers'. XOR compares the bits of the two numbers to see if they are identical or different. A binary 1 is returned if the two bits are different; a 0 is returned if they are identical:

First Number	Second Number	Bit Returned
1	1	0
1	0	1
0	1	1
0	0	0

The binary number returned is represented as an integer.

If 'number' is a real number, BASIC will convert it to an integer.

Examples

```
   PRINT XOR(72,32)
```

Prints the result, 104. The operation is performed on the binary representation of the two numbers:

```
          Integer                 Binary
                              Representation
            72                   01001000
            32                   00100000
           ---                  ----------
           104                   01101000
```

```
   PRINT XOR(104,32)
```

Prints 72:

```
          Integer                        Binary
                                     Representation
            104                         01101000
             32                         00100000
            ---                         ---------
             72                         01001000
```

```
   IF XOR(255,A) >= 128 THEN PRINT "SET BIT 8"
```

Performs the XOR operation on 255 and the value of A. If the condition is true, the statement is printed.

NOTE: See also OR and AND.

Sample Program

```
 80 REM     *** SAMPLE PROGRAM DEMONSTRATING XOR ***
 90 REM
100 PRINT "INPUT A LOWER OR UPPER CASE LETTER"
110 INPUT A$
120 B$ = CHR$(XOR(ASC(A$),32))
130 PRINT B$
140 GOTO 100
```

Radio Shack®

Section 3
BEDIT

Using BEDIT to Create and Edit BASIC Source Files.

TABLE OF CONTENTS
SECTION 3. BEDIT -- SOURCE PROGRAM EDITOR

* Note: The renumbering commands DO NOT RENUMBER LINE REFERENCES inside your program text; do not use them unless you are not concerned wth line references (GOTO, IF...THEN ..., GOSUB, etc.). To renumber your program properly, use the Compiler BASIC RENUMBER command.

INTRODUCTION

BEDIT lets you create and edit BASIC source files (the files that are input to the BASIC Compiler).

Capabilities and features:

. Allows you to load in ("chain") multiple source files.
. Single-key abbreviations for many commands
. Powerful intra-line editing mode like the edit mode in Model II Interpreter BASIC
. "M" command informs you of memory used/free at any time
. Global string find/change commands
. Editor provides line numbers in the range 0-65535
. Execute any TRSDOS library command without exiting the editor

SOURCE FILE FORMAT

Source files are written to disk in the format required by the BASIC compiler, as follows:

1. Files are variable-length record (VLR) type, as described in the TRSDOS Reference Manual, page 4/5.

2. Each record in the file corresponds to one line of source program. The first six data bytes (after the length-byte) in a record represent the line number in ASCII form followed by a blank space. The carriage return (<ENTER>) used to terminate the line during line insertion is not stored.

3. Text is stored exactly as it is displayed on the video, e.g., spaces are stored as spaces, not as a tab character.

4. No end-of-text code is stored in the data file.

TO START THE EDITOR

The editor program is included on the BASIC package diskette. It has the file name BEDIT.

To use the editor, put the BASIC diskette into one of your drives (drive 0 for single-drive users), and under TRSDOS READY, type:

```
    BEDIT
```

The editor will start up with the prompt:

```
    MODEL II BASIC EDITOR VERSION v.r
    OK
    >
```

Where v is the version and r is the release number. The > indicates you are in the command mode.

MODES OF OPERATION

There are three modes of operation:
COMMAND, for entering the editor commands
. INSERT, for entering your text lines
. EDIT, for interactive editing of a line of text

COMMAND MODE
The > prompt followed by the blinking cursor indicates the editor is waiting for you to type in a command. Every command must be completed by pressing <ENTER>. To cancel a command, press <ESC> or <BREAK>.

INSERT MODE
You enter text one line at a time; a line consists of up to 255 characters, including the five-digit line number provided by BEDIT. Line numbers can range from 0 to 65535.

The I command puts you in the insert mode. When you start inserting a line, the editor displays the five-digit line number followed by the blinking cursor. Your text can begin in column seven. (See the BASIC Language Reference Manual for column-field uses in BASIC source programs.)

To store the current line, press <ENTER>. The editor will display the next line number, and you can begin inserting into that line. To cancel the current line and return to the command mode, press <ESC> or <BREAK>. See I Command for details.

EDIT MODE
There are many powerful edit sub-commands--identical in most cases to those in Model II BASIC's Edit Mode. There is also a sub-edit insertion mode in which the keys you type are inserted into the line at the current cursor position.

To start editing a line, use the E command. After editing the line, press <ENTER> to save the corrected line and return to the command mode. To cancel all changes made and return to the command mode, press <Q>. For further details, see E Command.

USING THE COMMAND MODE

Special terms used in the command descriptions:

"text", "text buffer", "text area"
All refer to the BASIC source program currently in RAM.

"current line"
The line most recently inserted, displayed or referenced in a command. When there is no text in RAM, current line is set to 100. Immediately after a file is loaded, the current line is set to the beginning of the text.

"increment"
The value which is added to the current line number whenever the editor needs to compute a new line number. After startup, loading a new file, and when there is no text in RAM, the increment is set to 10.

"line-reference"
Either an actual line number from 0 to 65535, or one of the following special abbreviations:

Symbol	Meaning
#	Beginning line of text (lowest-numbered line)
.	Current line
*	Last line of text (highest-numbered line)

"line-range"
This can be either a single-line reference or a pair of line-references separated by a colon:

Sample Command	Meaning
P100	Prints line 100 only
P100:300	Prints all lines from 100 to 300
P#:.	Prints all lines from beginning to current

"delimiter"
A special character used to delimit (mark the beginning and end of) a string. Any of the following characters can be used:

! " # $ % & ' ( ) * + , - . / : ; < = > ?
Whichever character is used to mark the beginning of a string must also be used to mark the end of the string.

Sample use...	Marks this string...
'THIS " MARK'	THIS " MARK
/X'8000'/	X'8000'
&~~~~~~~&	~~~~~~~ (seven blanks)

(The "~" symbol represents a blank space. It is used only where necessary for emphasis or illustration.)

SPECIAL KEYS IN THE COMMAND MODE

<ESC> or <BREAK>
Press either key to cancel the command you are entering, or to abort a command which is currently being executed.

<TAB>
Advances the cursor to the next eight-column boundary (boundaries are at columns 8, 16, 24, ...)

<ENTER>
Pressing this key at the beginning of a command line displays the current line.

<up-arrow>
Pressing this key at the beginning of a command line displays the line which precedes the current line.

<down-arrow>
Pressing this key at the beginning of a command line displays the next line after the current line.

<-
Erases the command you are entering.

<HOLD>
Pauses H and P commands. Press any other key to continue.

COMMANDS

Note: Spaces are not significant in command lines. For example,

```
    P 1 : 5
```

has the same effect as

```
    P1:5
```

The P command is explained later on.

A

Enables automatic line-renumbering. Whenever a line-number collision occurs in the insert mode, the editor will automatically renumber the text lines, using the current line number as start-line and the current increment as increment. See the N commmand for details on renumbering.

Notes:
1. The renumbering commands DO NOT RENUMBER LINE REFERENCES inside your program text; do not use them unless you are not concerned wth line references (GOTO, IF...THEN ..., GOSUB, etc.). To renumber your program properly, use the Compiler BASIC RENUMBER command.

2. The A command does not put you in the insert mode; only the I and R commands do that. The A command simply sets an auto-renumbering "switch".

This function is disabled when you execute the N or L command.

B

Displays the beginning line (first line in the text area).

C/search-string/replacement-string/n

Finds, changes, and displays the first n lines that contain search-string. In each of these lines, search-string is changed to replacement-string. ONLY THE FIRST OCCURRENCE OF search-string IN A SINGLE LINE IS COUNTED AND CHANGED. If the end of text is reached before n finds, the message "SEARCH FAILS" will be displayed.

Upon completion of the command, the current line is set to the line of the last find, or to the first line of text when "SEARCH

FAILS" is displayed.

/search-string/ is a sequence of characters delimited by
a matched pair of characters from the set:

! " # $ % & ' ( ) * + , - . / : ; < = > ?

replacement-string/ is a sequence of characters terminated
by the same character used to delimit search-string.

n Tells the maximum number of "changes" you want. n can
be a number or an asterisk. The asterisk means change
and list all occurrences. If n is omitted, only the
first occurrence is changed and listed.

Sample Commands	Notes
C/VAR=/NET=/	Changes the first occurrence of "VAR=" to "NET=" in the first line that contains it.
C"VAR="NET="	Same as above.
C/RETRY/R/4	Changes the first occurrence of "RETRY" to "R" in the first four lines that contain it.
C/MISPELING/MIS-SPELLING/*	Changes the first occurrence of "MISPELING" to "MIS-SPELLING" in every line that contains it.
C/EXTRA//*	Changes the first occurrence of "EXTRA" to "" (null string) i.e., deletes the first "EXTRA" in every line that contains it.

D line-range

Deletes lines in the specified range. If line-range is omitted,
the current line is deleted.

Sample Commands	Notes
D. or D	Deletes the current line.
D2	Deletes line number 2.
D98:115	Deletes lines found in the range 98 to 115.
D1000:*	Deletes all lines numbered 1000 or higher to end of text.

E line-reference

Starts edit mode using the specified line. If line-reference is omitted, the current line is used.

Edit sub-commands:

<ENTER>	Ends editing and returns to command mode.
<F1>	Causes escape from sub-edit insertion (X, I, and H sub-commands) and returns to edit mode.
n <SPCBAR>	Advances cursor n columns. If n is omitted, 1 is used.
<BKSPC>	Backspaces the cursor one column without erasing.
L	"Lists" working copy of the line and starts a new working copy.
X	"Extends" line: positions cursor to end of line and enters sub-edit insertion mode. Use <F1> to escape to edit mode.
I	Enters sub-edit "insertion" mode at the current cursor position; use <F1> to escape. to edit mode.
A	("Again") Cancels changes and starts a new working copy of the line.
E	("End") Saves edited line and exits to command mode, > prompt.
Q	("Quit") Cancels changes and returns to command mode, > prompt.
H	"Hacks" remainder of line beginning at current cursor position and enters sub-edit insertion mode. Use <F1> to escape to edit mode.
nD	"Deletes" n characters beginning at current cursor position. If n is omitted, 1 is used. The deletion is not echoed; use <L> to see the line with characters deleted.
nC	"Changes" next n characters from the current cursor position, using the next n characters typed. If n is omitted, 1 is used.
nSc	("Search") Movse cursor to nth occurence of character c. Search starts at next character

	after the cursor. If n is omitted, 1 is used.
nKc	("Kill") Deletes all characters from current cursor position up to nth occurence of character c, counting from current cursor position. If n is omitted, 1 is used. The deletion is not echoed; use <L> to see the line with characters deleted.

F/search-string/n

Finds and displays the first n lines which contain search-string, starting at the current line. ONLY THE FIRST OCCURRENCE OF search-string IN A SINGLE LINE IS COUNTED. If the end of text is reached before n finds, the message "SEARCH FAILS" will be displayed.

Upon completion of the command, the current line is set to the line of the last find, or to the first line of text when "SEARCH FAILS" is displayed.

/search-string/ is a sequence of characters delimited by
    a matched pair of delimiters chosen from the set:

```
! " # $ % & ' ( ) * + , - . / : ; < = > ?
```

n   Tells the maximum number of "finds" you want. n can be a number or an asterisk. The asterisk means find and list all occurrences. If n is omitted, only the first occurrence is listed.

Sample Commands	Notes
F/VAR=/	Finds and displays the first line that contains the string "VAR=".
F"VAR="	Same as above.
F/RETRY/4	Finds and displays the first eight lines containing at least one occurrence of "RETRY".
F/MISPELING/*	Finds and displays every line containing at least one occurrence of "MISPELING".

H line-range

("Hard-copy") Lists to the printer all lines found in the specified range. If line-range is omitted, only the current line is printed.

The printer should be initialized (with FORMS) before you execute this command.

Sample Commands	Notes
H#:*	Lists all lines to the printer.
H7020	Lists line 7020 to the printer.
H672:800	Lists all lines found in the range 672 to 800.

I start-line, increment

Starts the insert mode.

start-line is a line-reference telling the editor where to begin inserting into the text. If omitted, the current line is used.

,increment is a number telling the editor how to compute successive line numbers. If omitted, the current increment is used.

If start-line is already in use, the editor will start with the next line number (start-line + increment).

Special Keys in the Insert Mode

<TAB>	Advances the cursor to the next eight-column boundary (8, 16, 24, ...).
<-	Erases the line and starts over.
<BKSPC>	Backspaces the cursor and erases the character.
<ENTER>	Marks the end of the current line. The editor will store the current line and start a new one, using increment to generate the next line number.

Line-Collisions
If the next line number is already in use (this is referred to as a "collision"), the editor will display the message:

```
    NO ROOM BETWEEN LINES
```

and return to the command mode. To allow further insertion at this point in your program, either renumber the text or try inserting with a smaller increment.

Note: If the automatic-renumbering function is enabled when a line collision occurs, the editor will renumber the text automatically before displaying the next line number. See A command.

Sample Commands	Notes
I	Start inserting at current line number, using current increment.
I,1	Start inserting at current line number, using 1 as an increment. If current line number is in use, start with current line plus 1.
I45,2	Start inserting at line 45 with an increment of 2. If line 45 is in use, start with line 47.
I100	Start inserting at line 100, using the current increment. If line 100 is in use, start with 100 plus increment.

L filespec

Loads a source file from disk. If there is already text in RAM, the editor will ask whether you want to chain the new text onto the end of the old, or clear out the old first.

filespec is a TRSDOS file specification for a VLR text file. The file may have been created by this BASIC editor or by another means. However, it must be in the BASIC source file format. (See Source File Format.)

Note: If you chain one file onto the end of another, it is up to you to make sure the are no line-number conflicts and that there are no duplicate line numbers.

The L command also disables the automatic renumbering function (see A command).

Immediately after chaining a file, you must use the N command to renumber the text. This will resolve any duplicate line numbers.

Sample Commands	Notes
L DEMO/BAS:1	Load DEMO/BAS from drive 1.
L XDATA	Load XDATA

M

Prints the number of characters in the source text (excluding the editor's line numbers) and the amount of memory free for text storage.

Sample Command	Notes
M	A typical response in a 64K system might might look like this: 000440- TEXT 046145- MEMORY Meaning you have 440 bytes of text, and 46145 free bytes of memory available.

N start-line,increment

Renumbers the entire text.

Note: The renumbering commands DO NOT RENUMBER LINE REFERENCES inside your program text; do not use them unless you are not concerned wth line references (GOTO, IF...THEN ..., GOSUB, etc.). To renumber your program properly, use the Compiler BASIC RENUMBER command.

start-line becomes the lowest line number when the text is renumbered. If start-line is omitted, the current line number is used.

increment is used in computing successive line numbers. If omitted, the current increment is used.

After renumbering, the current line is set to the highest line number in the renumbered text.

This command disables the automatic renumbering function (see A command).

Sample Commands	Notes
N	Renumbered text will start with current line; successive lines computed with current increment.
N100	Renumbered text will start with line 100; successive lines computed with the current value of increment.
N100,25	As above; line numbers at increments of 25.
N,100	Renumbered text will start with current

line number; line numbers at increments of 100.

P line-range

Prints the specified lines to the display. If line-range is omitted, 20 lines starting at the current line are displayed.

Sample Commands	Notes
P	Prints 20 lines starting at current line.
P233	Prints line 233.
P.	Prints the current line.
P*	Prints the last line.
P140:615	Prints the lines within the specified range. Lines 140 and 615 don't have to be existing line numbers.

Q

Terminates session and returns to TRSDOS. The source text is not written to disk.

R line-reference, increment

Replaces contents of the specified line and continue in insert mode. If line-reference is omitted, the current line is used. If increment is omitted, the current increment is used.

The R command is equivalent to the D (delete) command followed by the I (insert) command. When you enter the command, the editor deletes the specified line and puts you into the insert mode, starting with the line just deleted.
After you press <ENTER>, the editor will contine in the insert mode, prompting you to enter the text of the next line number. To escape from the insert mode, press <ESC> or <BREAK>.

Sample Commands	Notes
R125,3	Prompts you to insert replacement text for line 125. Subsequent line numbers will be generated with an increment of 3.
R*	Prompts you to insert replacement text for the highest numbered line in the text area; subsequent lines will be generated using the current increment.

S lib-command

Allows you to enter any TRSDOS library command, and return to the editor command mode upon completion. For a list of library commands, try the TRSDOS LIB command.

```
    Sample
    Command          Notes
    --------         -----
    S FORMS W=80     Executes the FORMS command, setting
                     the paper width to 80 characters. When
                     FORMS has completed execution, the
                     edit command prompt > is displayed.
```

W filespec

Writes the text in RAM into the specified file.

filespec is a TRSDOS file specification. If file already exists, its previous contents will be lost.

```
    Sample
    Commands         Notes
    --------         -----
    W DEMO/BAS:1     Save DEMO/CBL from drive 1.
    W XDATA          Save XDATA
```

X/search-string/replacement-string/n

This command is exactly like the C (Change) command, except that it displays the line to be changed and queries you (Change?) each time it finds search-string. If you answer Y, the line will be changed; any other answer leaves the line unchanged. In either case, the process continues until all first occurrences have been found.

```
    Sample
    Command          Notes
    --------         -----
    X/MISPELING/MSP/*
                     Changes the first occurrence of
                     "MISPELING" to "MSP"
                     in every line that contains it, but asks
                     you to confirm each change before it
                     is made.
```

Radio Shack®

Section 4
Programmers Information

Information on the Stand Alone Runtime System, Memory Usage, Assembly Language, Subprograms, and File Formats.

TRS-80 MODEL II

RSBASIC
PROGRAMMER'S INFORMATION
SECTION

JUNE 16, 1980

RYAN-McFARLAND CORPORATION
609 Deep Valley Drive
Rolling Hills Estates,
California 90274

TABLE OF CONTENTS

I. INTRODUCTION

This document contains all of the information required to compile, run and debug RSBASIC language programs on the Radio Shack TRS-80 Model II Microcomputer under the TRSDOS Operating System.

It assumes the reader is familiar with the RSBASIC Language, the general operation of the TRS-80 Model II Microcomputer, and the TRSDOS Operating System. The reader is specifically referred to:

TRS-80 Model II RSBASIC Language Manual
TRS-80 Model II Operation Manual
TRS-80 Model II Disk Operating System Reference Manual

This guide is organized such that each chapter fully describes a particular operational procedure. While the experienced user need only refer to the appropriate chapter, it is recommended that the first-time user read the complete guide prior to operation of the RSBASIC system.

II. OVERVIEW

RSBASIC operates on a TRS Model II Micro computer under the TRSDOS Operating System. It is actually two separate systems.

The full development system is used for editing, compiling, and checking out RSBASIC programs. The TRS Model II must be equipped with 64K bytes of memory to run the full development system.

The Stand-Alone Runtime system (RUNBASIC) is used for execution of previously compiled programs and execution and checkout of previously compiled programs whose resultant object programs require more memory than is available under the full development system. RUNBASIC will run on a TRS Model II with as little as 32K bytes of memory.

III. THE FULL DEVELOPMENT SYSTEM

The Full Development System consists of four modules: the Resident which always resides in memory, and three overlays:

1) The Editor,
2) The Compiler, and
3) The Runtime.

The Full Development System is entered via the RSBASIC command. The format is as follows:

```
    RSBASIC [filespec] [{T=nnnn,S=xxxx}]
```

where:

filespec is an optional RSBASIC source or object file which is to be run by the RSBASIC system. If filespec is omitted, the system prompts for input with an asterisk ('*').

T=nnnn indicates the highest memory address accessible to the RSBASIC system. The address nnnn is in hexadecimal notation.

S=xxxx indicates the system should reserve hexadecimal xxxx bytes for stack space. The default is &C0. This number should not be less than &20.

To exit the system, the SYSTEM command with no parameters is used. This will return control to the TRSDOS operating system.

The Editor

The Editor overlay is loaded by the Resident when editing functions are required.

The Editor allows manipulation of source programs. It is used to build the source programs which will be compiled and executed by the other parts of the system.

The Compiler

The Compiler is the heart of the RSBASIC System. It compiles the RSBASIC source statements into an interpretive object format which will be executed by the RSBASIC Runtime. Compilation proceeds from the beginning to the end of the program with any error information noted along the way.

There are four methods of invoking the Compiler. One is to issue the COMPILE command, specifying an input source file and an output object file. This method compiles the source program into object code one statement at a time and outputs the object code to the specified output file. The COMPILE command also allows the options of producing a listing of the source along with a cross-reference and memory-map. This listing can optionally be routed to the printer or, in a future release, to a disk file.

```
CO[MPILE][,]filespec, filespec [{LIST,MAP,PRT,XREF}]
```

The second method of invoking the compiler is to issue the RUN command with no parameters. This allows compilation and execution of the RSBASIC program currently in memory.

The third method is to issue the RUN command giving the optional filespec (RUN filespec). If 'filespec' specifies a source program, memory is cleared, the source program is read into memory, compiled, and executed. The 'filespec' may also specify an object program, in which case the compilation step is unnecessary.

The fourth method of invoking the compiler is to issue the STEP command. If necessary, this will compile the RSBASIC program in memory and allow the user to execute the resultant object code. The line number of the next line to be executed will be printed on the screen.

Control returns to the command mode following completion of a compilation, execution, or STEP.

The Runtime

The Runtime overlay is loaded to execute the RSBASIC object code in memory. It processes until one of the following occurs:

1) a user-defined breakpoint is reached, in which case a message is printed on the screen and control returns to the command mode.

2) when executing a STEP command, the start of the object code for the next (or the specified number) source line is reached, in which case a message is printed on the screen and control returns to the command mode.

3) a nonfatal error is detected, in which case an error message is printed on the screen and execution is continued.

4) a fatal error is detected, in which case an error message is printed on the screen, all open files are closed, and control returns to the command mode.

5) the program executes a STOP or END statement or executes the last statement of a program, in which case a stop message is printed on the screen, all open files are closed and control returns to the command mode.

Program Debug

In order to enhance program development, a debug facility is provided. Debug is initiated in one of three ways:

1) The STEP command,
   STEP

2) The BREAK command,
   BREAK line number, line number,... .

3) The TRACE command,
   TRACE ON/OFF

The STEP command allows the user to execute his program one or more lines at a time. After each step, control returns to the command mode to allow the user to input new debug commands. Debug is complete when either the STOP or END statements have been reached or the GO command is issued.

The BREAK command is used to set breakpoints at various lines within the program. Execution is initiated with the GO command and proceeds until either a breakpoint is reached or the STOP or END statements have been executed. Control is again returned to the command mode.

The TRACE command is used to produce a trace line of each line number executed. TRACE may be used in conjunction with other debug commands. The format of the TRACE line is

```
     LINE nnnn
```

where nnnn is the line number of the next line to be executed.

When control has returned to the command mode, the remaining debug command may be used, the DISPLAY command:

```
DI[SPLAY] [[routine name];]variable, [[routine name];]variable...
```

where:

routine name describes the routine where the variable resides. Complete descriptions of all debug commands may be found in the RSBASIC Language Manual.

IV. THE STAND-ALONE RUNTIME SYSTEM

The Stand-Alone Runtime System is a single module system which interprets object code from previously compiled RSBASIC source programs. It is invoked with the RUNBASIC command and processes in much the same manner as the Full Development System Runtime. The Stand-Alone Runtime System debugging facility, however, differs in that only breakpoints may be set; there is no STEP facility. At a breakpoint data items may be displayed to checkpoint program accuracy.

Format of the RUNBASIC command:

```
    RUNBASIC filespec [{D,B,T=xxxx, S=nnn}]
```

where:

D causes the system to load and execute with interactive debug.
T = xxxx reserves memory above hexadecimal address xxxx for user subroutines. (default is TOP)
B enables the BREAK key for halting execution (default is disabled)
S = nnnn reserves hexadecimal nnnn bytes for the runtime stack. (default is &C0)

The options may appear in any order.

STAND-ALONE DEBUG

The commands to the Stand-Alone Debug module are much the same as the corresponding commands to the Full Development System. Since the symbol table is not available to the debug module, locations corresponding to the listing generated by the compiler are used to denote both line numbers in the BREAK command and variables in the DISPLAY command.

Real and integer scalars in the common area are denoted by a single quote after the location just as they are on the Symbolic Memory Map; i.e., 01A' is location 01A in the common area. An asterisk before the location is used to denote formal parameters to subroutines; i.e., *0347 is used to display the current contents of the formal parameter at location 0347. Note that a leading 0 is needed on the location when the leading hexadecimal digit is A through F to be sure the debug module does not mistake it for a subprogram name.

If the D option is chosen, debug will prompt for a command under the following circumstances:

1) after the program to be run is loaded into memory, but before execution begins.

2) after a message is printed on the screen detailing the filespec specified in a CHAIN statement and where the statement occurred.

3) after loading the program specified in a CHAIN statement, but before execution begins.

4) after any fatal error message is printed on the screen.

5) after normal termination of the program.

At any of the above points, any debug command may be entered, however, at points 4) and 5), the GO command and the SY command without a parameter will both cause a return to the TRSDOS READY mode.

STAND-ALONE DEBUG COMMANDS

All commands to the debugger are two characters only; anything else results in a COMMAND SYNTAX ERROR.

BREAKPOINT Command   BR <address>,...

The breakpoint command will cause execution of the RSBASIC program to be suspended when the instructon at <address> is reached.

If not qualified, <address> refers to the "current" program or subprogram; that is, the program in which execution was suspended by the breakpoint. Before execution begins, the current program is defined as the main program.

A semicolon before the <address> forces it to be relative to the main program, while a subroutine name before the semicolon forces the <address> to be relative to that subroutine.

The breakpoint command only (not followed by <address>) clears all breakpoints previously set.

DISPLAY Command

DI <address>,...

The display command formats the current contents of a variable according to its type and prints it. The <address> is that location corresponding to the desired variable on the Symbolic Memory Map generated by the compiler.

An unqualified <address> defaults to that program in which execution was suspended, or the main program if execution has not begun. A semicolon before the <address> forces it to be relative to the main program, while a subroutine name before the semicolon forces the <address> to be relative to that subroutine.

Type information is conveyed by the characters "%" and "$" appended to <address>. The type defaults to real. An array element may be displayed by appending the subscripts in parenthesis to <address>. Subscripts must be integer constants.

For Example:

```
      DI SUB1;*0304$(1,1),;0306%
```

The above command will display the current contents of the string array element in the first row and first column of the two-dimensional string array which was passed as the formal parameter at location 0304 to subroutine SUB1, followed by the integer variable at location 0306 in the main program.

DUMP Command

DU <address 1>[-<address 2>]

The dump command is used to dump memory as hexadecimal bytes. The qualification of <address 1> is the same as for the breakpoint command.

GO Command

GO

The go command either begins execution or resumes after a breakpoint is reached.

__SYSTEM Command__   SY ["TRSDOS System Command"]

The system command passes a string to TRSDOS as if the string were entered in response to the TRSDOS READY prompt. If no string is given, control is returned to TRSDOS.

V. MEMORY USAGE AND DATA STORAGE

<u>Object Program Structure</u>

RSBASIC programs use two distinct storage areas: PSECT for storage of instructions, constants, addresses, and dope (array and string descriptors), and DSECT for storage of all variable data. The system will allocate both these sections within its controlled memory area as follows:

```
-------------------------
!     COMMON Storage    !
!        (If any)       !
+-----------------------+
!     MAIN ROUTINE      !
!     PSECT Storage     !
+-----------------------+
!     MAIN ROUTINE      !
!     DSECT Storage     !
+-----------------------+
!           .           !
!           .           !
!           .           !
!                       !
+-----------------------+
!     SUBROUTINE N      !
!     PSECT Storage     !
+-----------------------+
!     SUBROUTINE N      !
!     DSECT Storage     !
+-----------------------+
!     ADDRESS TABLE     !
-------------------------
```

Storage of Integers*

Integers are stored in 16-BIT two's complement form. The least significant byte is stored in the first memory byte and the most significant in the second. The examples below illustrate this storage format.

Storage of +5 at hex address 00A1:

```
              7 6 5 4 3 2 1 0
             -----------------
00A1         ! 0 0 0 0 0 1 0 1 !
00A2         ! 0 0 0 0 0 0 0 0 !
             -----------------
```

Storage of -5 at hex address 0073:

```
              7 6 5 4 3 2 1 0
             -----------------
0073         ! 1 1 1 1 1 0 1 1 !
0074         ! 1 1 1 1 1 1 1 1 !
             -----------------
```

(-5=(COMPLEMENT OF +5)+1)

The numbers which may be thus represented are the integers in the range

-32768 TO +32767

This, therefore, defines the range of integers in the RSBASIC system.

*For more information on the storage of integers in two's complement form, see "TRS80 Assembly Language Programming" by Bill Barden, Jr., Radio Shack Catalog Number 62-2006.

Storage of Decimals

Decimals are stored in 8 bytes with the first byte containing the sign and exponent and the remaining 7 bytes containing 14 binary coded decimal digits representing the mantissa.

The first bit of the first byte is the sign. A 0 bit denotes a positive number and 1 bit denotes a negative number. The other 7 bits represent a biased binary exponent of ten. The exponent is biased by &40. That is, an exponent of &40 is equivalent to 0.

The mantissa is normalized to the left. This means the first digit of the mantissa is zero only if the number is zero. The exponent is adjusted accordingly. An assumed decimal point is to the left of the mantissa.

The examples below illustrate this storage format.

Storage of 5.6 at hex address 00A1:

```
      -------------------------------------------------
00A1  | 4 1 | 5 6 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 |
      -------------------------------------------------
```

Storage of -5.6 at hex address 00A9:

```
      -------------------------------------------------
00A9  | C 1 | 5 6 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 | 0 0 |
      -------------------------------------------------
```

Storage of 2.368714E10 at hex address 00B1:

```
      -------------------------------------------------
00B1  | 4 B | 2 3 | 6 8 | 7 1 | 4 0 | 0 0 | 0 0 | 0 0 |
      -------------------------------------------------
```

This is equivalent to

```
        .2368714 X 10**(75-64)

        or .2368714 X 10**(11)
```

The numbers which may be thus represented are the real numbers in the range

```
             -0.99999999999999*10^+63 to -0.99999999999999*10^-64
         and +0.99999999999999*10^-64 to +0.99999999999999*10^+63
```

Storage of Numeric Arrays

Arrays of numbers are stored in memory by row with each number occupying two bytes for integer and eight bytes for decimal. The storage of single and double dimensioned arrays is illustrated in the two diagrams below:

Single dimension integer array A% with 3 members starting at hex address 0132:

```
0132   ----------
0133   ! A%(0)  !
0134   ----------
0135   ! A%(1)  !
0136   ----------
0137   ! A%(2)  !
       ----------
```

Double dimensioned integer array B% with 3 rows (first subscript) and 2 columns (second subscript) starting at hex address 3EB7:

```
3EB7   -----------
3EB8   ! B%(0,0) !
3EB9   -----------
3EBA   ! B%(0,1) !
3EBB   -----------
3EBC   ! B%(1,0) !
3EBD   -----------
3EBE   ! B%(1,1) !
3EBF   -----------
3EC0   ! B%(2,0) !
3EC1   -----------
3EC2   ! B%(2,1) !
       -----------
```

As can be seen from the examples above, the address of an element in a single dimensioned array is

```
ARRAY BASE + S*(SUBSCRIPT)
```

while the address of an element of a double dimensioned array element is

```
ARRAY BASE+S*((MAX SUBSCRIPT2+1)*SUBSCRIPT1+SUBSCRIPT2)
```

where S is either 2 for integer or 8 for decimal. For instance,

```
          AX(1) above would be:

               0132+2*(1)=0134

          BX(1,0) above would be:

               3EB7+2*((1+1)*1+0)=3EBB
```

The single dimensioned array can be thought of as a special case of the double dimensioned array with a MAX SUBSCRIPT2 of -1 if its subscript is treated as "SUBSCRIPT2". This implies that in each subscript calculation, two constants will be required -- the ARRAY BASE and MAX SUBSCRIPT2. MAX SUBSCRIPT1 is also needed for subscript checking.

For each array in the RSBASIC system, these three constants are stored in a memory block referred to as the array dope. In the example below, the array dope for the two example arrays is shown.

```
     Array Dope for A% and B% above
     Dope begins at hex address 1A75

                     -------
A%  Dope 1A75        | 3 2 |     A%  Base
         1A76        | 0 1 |
                     -------
         1A77        | 0 2 |     A%  Max Subscript1
         1A78        | 0 0 |
                     -------
         1A79        | F F |     A%  Max Subscript2
         1A7A        | F F |
                     -------
         1A7B        | 0 0 |     Array type (0=integer,1=real)
         1H7C        | 0 0 |     not used
                     -------
B%  Dope 1A7D        | B 7 |     B%  Base
         1A7E        | 3 E |
                     -------
         1A7F        | 0 2 |     B%  Max Subscript1
         1A80        | 0 0 |
                     -------
         1A81        | 0 1 |     B%  Max Subscript2
         1A82        | 0 0 |
                     -------
                     | 0 0 |     Type (integer)
                     | 0 0 |     not used
                     -------
```

Storage of Strings

Strings are stored one ASCII character per byte. The current length of the string in bytes is stored in a one-byte binary field at the start of the string. The examples below show how this works.

```
          "HELLO" stored at hex address 0175

                   -------
          0175     | 0 5 |     Current Length
                   -------
          0176     | "H" |
                   -------
          0177     | "E" |
                   -------     Current Value
          0178     | "L" |
                   -------
          0179     | "L" |
                   -------
          017A     | "O" |
                   -------


          String Variable C$, Max Length=10
          Starting at hex address 268A
          Current value is "BASIC"

                      -------
          C$  268A    | 0 5 |   C$ Current Length
                      -------
              268B    | "B" |
                      -------
              268C    | "A" |
                      -------
              268D    | "S" |   C$ Current Value
                      -------
              268E    | "I" |
                      -------
              268F    | "C" |
                      -------
              2690    |  x  |
                      -------
              2691    |  x  |
                      -------
              2692    |  x  |   C$ Currently Unused
                      -------
              2693    |  x  |
                      -------
              2694    |  x  |
                      -------
```

Strings may be empty, i.e., they may have a current length of 0, or they may have any length up to and including their declared maximum. For each declared string, a total of MAX LENGTH+1 bytes is reserved for the storage of the string and its current length.

During program operation, the MAX LENGTH of a string variable will be required to control storing operations into the string. Thus, for string variables, two constants are required during program operation -- the STRING ADDRESS as well as the MAX LENGTH.

For each string variable, these constants are stored in a memory block called the string dope. In the example below, string dope is shown for the example string C$.

```
          String Dope for C$
          Dope begins at hex address 2BC1

                              -------
          C$ DOPE  2BC1       | 8A  |   C$ Address
                              -------
                   2BC2       | 26  |
                              -------
                   2BC3       | 0A  |   C$ Max Length
                              -------
```

Storage of String Arrays

Strings may also be stored in single or double dimensioned string arrays in which each element has the same maximum length but may, of course, have unique current value and length. The example below shows the storage of a single dimensioned string array A$ having three elements each with a maximum length of 5 characters:

```
          String Array A$, Max Length=5, 3 elements
          Starting at hex address 75A3

          A$(0)="HELLO", A$(1)="FROM", A$(2)="RMC"

                                 -------
          A$(0)  75A3            ! 0 5 !   A$(0) Current Length
                                 -------
                 75A4            ! "H" !
                                 -------
                 75A5            ! "E" !   A$(0) Current Value
                                 -------
                 75A6            ! "L" !
                                 -------
                 75A7            ! "L" !
                                 -------
                 75A8            ! "O" !
                                 -------
          A$(1)  75A9            ! 0 4 !   A$(1) Current Length
                                 -------
                 75AA            ! "F" !
                                 -------
                 75AB            ! "R" !   A$(1) Current Value
                                 -------
                 75AC            ! "O" !
                                 -------
                 75AD            ! "M" !
                                 -------
                 75AE            !  x  !   A$(1) Currently Unused
                                 -------
          A$(2)  75AF            ! 0 3 !   A$(2) Current Length
                                 -------
                 75B0            ! "R" !
                                 -------
                 75B1            ! "M" !   A$(2) Current Value
                                 -------
                 75B2            ! "C" !
                                 -------
                 75B3            !  x  !   A$(2) Currently Unused
                                 -------
                 75B4            !  x  !
                                 -------
```

Item order of double dimensioned string arrays is the same as for double dimensioned numeric arrays.

The address of a single dimensioned string array element is calculated as follows:

```
          STRING ARRAY BASE+(MAX LENGTH+1)*(SUBSCRIPT)

                  e.g., for A$(1) above:

                     75A3+(5+1)*(1)=75A9
```

The address of a double dimensioned string array element is calculated as follows:

```
         STRING ARRAY BASE+(MAX LENGTH+1)*
                 ((MAX SUBSCRIPT2+1)*SUBSCRIPT1+SUBSCRIPT2))
```

Dope for string arrays is similar to dope for arrays of numbers. The first two bytes are the STRING BASE, followed by two bytes for MAX SUBSCRIPT1, followed by two bytes for MAX SUBSCRIPT2 (-1 if single dimensioned), followed by a one-byte array type (02 for string), followed by a one-byte MAX LENGTH.

In the example below, string dope is shown for the example single dimensioned string array A$.

```
          String Dope for A$
          Dope begins at hex address 2BC4

                                --------
          A$ Dope  2BC4         ! A3   !   A$ Address
                   2BC5         ! 75   !
                                --------
                   2BC6         ! 02   !   A$ Max Subscript 1
                   2BC7         ! 00   !
                                --------
                   2BC8         ! FF   !   A$ Max Subscript 2
                   2BC9         ! FF   !
                                --------
                                ! 02   !   Array Type
                                ! 05   !   A$ Max Length
                                --------
```

Stack Usage

An RSBASIC program uses the stack for storing return addresses and the state of subroutines.

Each GOSUB and function call (DEF function) uses two bytes.

Each CALL to an RSBASIC external subroutine uses 10 bytes.

The system uses about 32 bytes for internal storage.

To calculate the expected stack size, estimate the maximum number of nested gosubs, function calls, and subroutines that could occur in a program. The stack size should be 2*(number of nested gosubs and function calls) + 10*(number of nested subroutines) + 32.

For example, a program which could nest to a depth of 80 gosubs would require a stack size of &C0 bytes.

The system checks for stack overflow and for RETURN's without a matching GOSUB at execution. The size of the stack is determined by the S option in both RUNBASIC and RSBASIC. The default is &C0 bytes.

VI. ASSEMBLY LANGUAGE SUBPROGRAMS

Assembly language subprograms may be called by RSBASIC programs. However, the user is responsible for loading them by use of the TRSDOS LOAD command into memory locations which do not conflict with the RSBASIC system and for protecting them from overwrite by the RSBASIC system via the T (top of memory) parameter on the RSBASIC and RUNBASIC commands.

Setup

Calling an assembly language subprogram from an RSBASIC program requires the same statement format as a normal RSBASIC subprogram call. However, since the RSBASIC system will not know where the user's assembly language program is loaded, this information must be supplied via the EXT statement in the format;

```
    ln EXT subname = XXXX,...
```

where:

subname is the subprogram's name as used in CALL's of the subprogram, and XXXX is the address where it has been, or will be, loaded.

Parameter Passing

Upon entry to the user's assembly language subprogram information from the RSBASIC system is passed as follows:

```
    (SP) ---> the return address*

    BC ---> the calling routines parameter list (if any),

    DE ---> a parameter decoding routine for use in retrieving
            subroutine parameter addresses and types.
```

*Note: The Runtime requires that information currently on the stack other than the return address must not be altered and must remain in its relative position.

In order to pick up any parameter addresses, the routine referenced in DE must be 'called'. Since this routine has saved all pertinent parameter information, it requires no parameters; however, it returns the following:

```
B= argument type, 0 for integer scalar
                  1 for real scalar
                  2 for string scalar
                  4 for integer array
                  5 for real array
                  6 for string array

DE = argument address (for string scalars, this is the
                       address of the string dope, for
                       arrays, this is the address of
                       the array dope)

A = return code,  0 for argument returned
                 -1 for no more arguments
```

Care must be taken when passing parameters back to the RSBASIC program to ensure that their formats are correct (see Storage of Data section).

<u>Returning to RSBASIC</u>

At completion of an assembly language subprogram, return is made to the calling program by passing control to the address which was pointed to by the stack pointer.

VII. THE RSBASIC FILE SYSTEM AND FILE FORMATS

System Supported Files

Three types of files are supported in RSBASIC: sequential, direct (random), and indexed sequential (ISAM).

Files are specified in the user's program in a manner consistent with the TRSDOS filespec, of the form

filename/ext.password:d(diskette name)

where:

'filename' is required.

'/ext' is an optional name-extension.

'.password' is an optional password. When omitted no password checking is performed.

':d' is an optional drive specification. When omitted the system does an automatic search, starting with drive 0.

'(diskette name)' is optional. When omitted no disk name checking is performed.

Sequential Files

Sequential files are created by Runtime as either variable length or fixed length records, according to user specification (i.e., if a LENGTH parameter is supplied in the OPEN statement, the records will be fixed length; otherwise, they will be variable length). If the file exists at OPEN time, the file type and record length are used as defined by TRSDOS.

Sequential files do not allow DELETE or Update. The maximum record length for sequential files is 255 bytes.

Direct Files

Direct files are fixed length record (FLR) files. They differ from standard TRSDOS Direct files in that appended to the front of each record is a two-byte record length. The maximum record length for direct files is 254 bytes.

Indexed Files

Indexed (ISAM) files may be referenced in either the sequential or random mode. Each record in an indexed file is uniquely identified by the value of the associated key. In RSBASIC, the key need not be part of the data written in the file. It is used as a roadmap in order to retrieve the record on which the data are stored.

The RSBASIC single-key ISAM structure is built on a TRSDOS direct file with 256-byte physical records. Internally, the ISAM module uses 32-byte logical records called allocatable units (AU's).

There are four types of objects in an ISAM file:

1) Header (1 AU)
2) Tree   (each node = 16 AU's)
3) Linked Lists
4) User data records

The file header starts at AU 1 (the first). There is only one tree in which all key values are maintained. The header contains a pointer to the key tree's root node. The header also contains pointers to the start of two free lists. These two lists contain free directory (tree) nodes and free user records. Directory nodes contain pointers which point to the associated data record.

When a new object (node or data record) needs to be created, an entry on one of the free lists is reused if one exists. Otherwise, space is allocated at the current end of file. Variable length data are stored in fixed length data records to allow space to be recovered more easily.

The physical format of the header, a node record, and a user data record are as follows:

```
Header:  header code word
         # of AU's to store header (1)
         # of AU's to store data record (m)
         head of free node list
         number of free nodes
         head of free record list
         number of free records
         head of free duplicate block list (0)
         number of free duplicate blocks (0)
         next free AU
         flag word
         # of keys (1)
         key size
         key offset (0)
         tree height
         root of index tree
         next available stamp # (0)

Node:    node count word
         number of keys in this node
         left pointer
        -------------
        ! data pointer
        ! key value
        ! right pointer
        -------------
              .
              .
              .

User Data Record:  byte count
                  ------------
                  ! data byte
                  ------------
                       .
                       .
                       .
```

Indexed records are 'mapped' onto direct file records of 256 bytes (standard TRSDOS sector size) regardless of their actual size.

The formula shown below should approximize the number of 256 byte sectors that a given file will require on disk. The actual number of granules is this number divided by 5.

```
#Sectors =   1 + INT(1 + R* INT((S + 33)/32)/8)

           + INT(1 + 2*R/INT(252/K + 8)

where:  INT = Integer value

          R = Number of records in the file

          S = Size of largest record (in bytes)

          K = Size of key field (in bytes)

Example: 1000 records i file (R = 1000);
         max record size is 190 bytes (S = 190);
         key is 6 bytes (K = 6)

#Sectors:   1 + INT(1 + 1000*INT((190 + 33)/32)/8)

          + INT(1 + 2*1000/INT(252/(6 + 8))

          = 1 + 751 + 112 = 864
```

<u>RSBASIC File Formats</u>

Within the system file structure RSBASIC supports three subfile systems which can be mapped over any of the three system file formats:

1) Free Format,
2) USING Format, and
3) Binary Format.

Free Format files are constructed to resemble an RSBASIC program input stream with trailing zeros and blanks deleted and items separated by commas. All items are in ASCII format, so that an INPUT operation from such a file differs from console input only in the fact that input comes from a diskette file.

USING format files are in ASCII format, but items are not separated by commas; rather, they are set into a string structure as dictated by the elements of the USING string specified when the file was written.

Binary Format files, unlike the others, are constructed in internal format in the following method:

1) integers are output as two-byte binary numbers;

2) decimals are output in their internal format with trailing zeros truncated and with a leading one-byte length count;

3) strings are output as a one-byte count followed by their ASCII representation minus trailing blanks.

The whole record is then output with a one-byte record length count in front.

RSBASIC, RSCOBOL, and ISAM Files

The format of the RSBASIC indexed sequential (ISAM) file was designed to provide a method by which an RSBASIC program and an RSCOBOL program may communicate. By adhering to a few simple rules, the RSBASIC programmer may successfully read, write and update an ISAM file created by RSCOBOL. The rules are simple but quite stringent for both RSCOBOL and RSBASIC. If any of them are ignored, the data in the file may be irretrievably lost.

1) The file must be single-key only

   RSBASIC language syntax only permits one key

2) The key must be written as part of the data record

   RSBASIC ISAM format does not require this, but RSCOBOL does.

3) The records must be fixed-format ASCII

   RSCOBOL has provision for neither binary data nor variant records. The easiest way for an RSBASIC programmer to ensure this is with the PRINT USING and INPUT USING statements. The Image used is analogous to the RSCOBOL record descriptor.

If the RSBASIC ISAM file is not to be accessed by an RSCOBOL program, the above rules do not apply and any of the RSBASIC I/O statements may be employed.

Notice that in RSBASIC the record is padded on the right with blanks or zeroes, as appropriate for the record type (ASCII or binary, respectively).

Sequential reading of an ISAM file is possible in RSBASIC by simply not specifying a KEY on the INPUT or READ statement. The record input will be the one whose key is next in the ASCII collating sequence. The value of the KEY last read will be assigned as the output of the KEY$ function.

Radio Shack®

Appendix

ERROR MESSAGES AND RETURNS

Resident Error Messages

OVERFLOW

The system has exhausted its available memory space.

If overflow occurs during an APPEND, then none of the new lines are appended. During the OLD, lines are included up to the point where overflow occurred. During RENUMBER, all lines are renumbered but references to line numbers are updated only up to the point where overflow occurred.

SYNTAX

Improper command, redundant information following command, or improperly formed number or name.

PARAMETERS

Improper parameters have been included in the RSBASIC initiation command line.

Editor Error Messages

AUTO

Incorrect specification of the AUTO command.

CHANGE

Incorrect parameter specification in the CHANGE command.

DUPLICATE

Execution of the DUPLICATE command as specified would overwrite an existing program line.

FILE FORMAT

An attempt was made to load a file which was not an object file or was improperly formatted. May occur during a CHAIN or LOAD.

LINE NUMBER

Line number specification or line number range is incorrect.

RENUMBER

A renumber operation (RENUMBER or APPEND) has been requested which would generate a line number larger than 65535 or the increment is zero.

SYNTAX

Improper command, redundant information following command, or improperly formed number or name.

Compiler Error Messages

Compiler error messages, when appropriate, will print a '$' character under the item in the line which prompted the error. Error messages will be printed under the line in which the error occurs.

COMMON SIZE

There exists a discrepancy in the COMMON SIZES between a main and subprogram.

COUNT

Inconsistant number of arguments in a subprogram or function call.

DOUBLE DEFINITION

Variable or array has already been declared in a SUB or DIM statement and may not be declared again.

FILE FORMAT

An input file is not in the expected format.

FILE UNAVAILABLE -- TRSDOS ERROR XX

The file specified for input or output cannot be accessed. XX = TRSDOS error number.

LOGICAL EXPRESSION EXPECTED

An invalid specification of a logical expression has been detected.

NUMERIC OR STRING EXPRESSION EXPECTED

A logical expression has been detected where a numeric or string expression was syntactically expected. For example,

```
10 A=B OR C.
```

OVERFLOW

Scalar or Array offsets have exceeded &FFFF.

ORDER

SUB must be the first active statement of a subprogram. DEF, COM, REAL, INTEGER and STRING must precede executable statements: FOR must precede NEXT; SUB may be preceded only by END. Or, FOR loops may be nested but must not overlap.

REFERENCE

Programs may not CALL themselves. String valued functions or string expressions may not be used as arguments in function references or subroutine CALLS. Arrays may not appear in function references, expressions, assignments, or relations -- only subroutine CALLS.

SIZE

Specification of a size limit, dimension, or value which exceeds allowable storage capacity.

SUBPROGRAM

SUBEND may appear only at the end of a subprogram.

SYNTAX

Improperly formed expression or incorrect punctuation. Redundant information at end of statement. Missing or misspelled keyword such as TO, THEN, GOSUB, or GOTO. Improperly formed name. Improperly formed string or numeric constant.

TYPE

Strings and numbers may not be mixed in arithmetic expressions. The type of a variable does not agree with its use in the current context.

UNCLOSED FOR LOOPS

LINE NUMBER nnnn WITH INDEX VARIABLE name

UNDEFINED

A referenced function or variable has not been defined.

WARNING: TYPE

An invalid type has been specified in a function call. Corrective action has been taken.

Runtime Error messages

Runtime error messages are of the format:

    message text ERROR LINE ####.

There are two types of Runtime errors: fatal and nonfatal. Fatal errors cause immediate cessation of execution; nonfatal errors resume processing after a message of the error has been displayed.

The number in parenthesis is the error number returned by the ERR function.

Fatal errors are:

(01) END OF FILE

Read attempt at end of file.

(02) IO PARAMETER

The parameters of an I/O statement are not recognized.

(03) COMPILATION

The program contains a compilation error.

(04) USING

A PRINTUSING or INPUTUSING statement has attempted to print or input data using an Image which contains no format specifications.

(05) INPUT SYNTAX

Invalid type of data received on an INPUT statement.

(06) BUFFER SIZE

Record length for a file is less than zone size for standard format print.

(07) OUT OF DATA

An attempt was made to READ past the end of the DATA list.

(08) READ DATA TYPE

There is a type discrepancy between the variable data requested and that of the DATA list.

(09) UNDEFINED REFERENCE

A reference has been made to an unknown line number or external routine.

(10) SUBSCRIPT

A subscript is out of range.

(11) ARGUMENTS

The number, type, or value of arguments in an I/O statement or subroutine call does not match the corresponding file record or subroutine parameter list.

(12) RETURN

A RETURN has been executed with no matching GOSUB.

(13) OVERFLOW

The stack memory has been exhausted due to excessive GOSUB and/or CALL nesting.

(14) INVALID UNIT

An invalid or undefined unit number has specified in an I/O statement.

(15) UNIT NOT OPEN

An I/O statement refers to a unit which has not been opened.

(16) UNIT OPEN

Attempted OPEN of an already open unit.

(17) FILE DCB SPACE EXHAUSTED

An attempt has been made to open more units than can be accommodated at one time, due to either system or memory limitations.

(18) INVALID FILESPEC

A filespec has been invalidly specified.

(19) KEY LENGTH

A key length less than one or greater than 127 has been detected.

(22) BINARY READ

Input data does not match the READ list.

(23) BINARY WRITE

Output data does not fit in a record.

(24) DELETED RECORD

Attempted READ of a deleted binary record.

(25) INVALID KEY

The ISAM processor has detected an illegal key value.

(26) KEY BOUNDARY

The ISAM processor has detected an invalid key boundary within an existing ISAM file.

(27) RECORD POINTER

The ISAM processor has detected an invalid record pointer within an existing ISAM file.

(28) INVALID

The ISAM processor has detected an invalid index within an existing ISAM file.

Nonfatal errors are:

(30) INPUT SIZE

A value greater than can be accomodated in the specified variable has been input. The data item is set to the maximum value and the specified sign is set to the maximum value and the specified sign.

(31) OUTPUT SIZE

Numeric value is too long for the Image specification. Field is filled with #. No message is printed unless the error is produced by ERROR statement.

(32) NUMERIC OVERFLOW

Overflow during expression evaluation. Sets value to maximum value with algebraically correct sign and continues.

(33) NUMERIC UNDERFLOW

Underflow during expression evaluation. The value is set to zero. Occurs only on decimal arithmetic.

(34) DIVISION BY ZERO

The value is set to the maximum for the type.

(35) SQR

Attempt to find the square root of a negative number. The value returned is the square root of the absolute value of the input number.

(36) LOG

Attempt to find the LOG of zero or a negative number. For zero the result is set to the maximum negative value. For a negative number the result is set to the LOG of the absolute value.

(37) POWER

A negative number is raised to a nonintegral power or zero raised to a negative power. Results are minus the power of the absolute value and maximum value, respectively.

Converting Data Files

You cannot read Interpreter BASIC data files directly using Compiler BASIC's file input statements. This is because the two BASICs use different record storage formats.

Therefore we have provided a set of Z-80 routines (all contained in the file "DOSFILES") which can can be used by your Compiler BASIC program to read Interpreter BASIC data records. In fact, these routines can be used by your Compiler BASIC program to read any TRSDOS data files.

There are three routines available. We will prefix a "$" to each name to remind you that they are Z-80 routines, not BASIC statements.

$OPEN	Opens the data file.
$GETREC	Reads a record.
$CLOSE	Closes the data file.

The routines allow you to input data records. It is up to your program to:

. Locate the data fields
. Make any data conversions that may be required*
. Save the data in a Compiler BASIC data file

* In Direct Access files, Interpreter BASIC uses binary floating point format for single- and double-precision numbers. Compiler BASIC uses a binary-coded decimal format for all REAL (non-INTEGER) numbers.

We suggest you use Interpreter BASIC to input such data records (using FIELD/GET/CVD,CVS and CVI) and then output them in ASCII format using PRINT #. This will allow Compiler BASIC to read in the data fields correctly.

Demonstration Program

The RSBASIC diskette contains a demonstration program and data file, SAMPFILES/BAS and FILE. SAMPFILES/BAS is a Compiler BASIC source program which reads in data records from the file named FILE. To try out SAMPFILES/BAS, do this:

```
    TRSDOS READY
    RSBASIC {T=E9FF}
    OLD SAMPFILES/BAS
    LIST
```

Take a look at the program listing. There are several explanatory REMarks in it. Then run it:

```
    RUN
```

The program will read in the records and display them. If the data file contains non-displayable data, you will get a TRSDOS ERROR message; but our sample data file contains only displayable information.

To Use the DOSFILES Routines

1. You must know the data file type and record length. Interpreter BASIC data files are always "F". Interpreter BASIC sequential-access data files have record length 1; direct-access files have a record length from 1 to 256.

If you don't know the file type or record length, look at the TRSDOS DIRectory entry for that file.

3. Start RSBASIC with memory protection set at X'E9FF', like this:

```
    TRSDOS READY
    RSBASIC {T=E9FF}
```

4. Load DOSFILES with a BASIC statement like this:

```
    ln SYSTEM "LOAD DOSFILES"
```

where 'ln' is the program line number.

5. Set up the RSBASIC EXTernal calls to the Z-80 routines with a BASIC statement like this:

```
    ln EXT OPEN=&EA01, GETREC=&EA04, CLOSE=&EA07
```

6. OPEN the data file by CALLing the external subroutine:

```
    ln CALL "OPEN"; file, reclen, type, err
```

where:

'file' is a string variable or constant containing the TRSDOS file specification.

'reclen' is an integer variable or constant containing the record length.

'type' is a string variable or constant containing the file type ("F" or "V").

'err' is an integer variable which will contain an error code upon return from the external subroutine.

Note: You do not provide the values for 'reclen', 'type', and 'err'. These are provided by the external subroutine.

Upon return from $OPEN, your BASIC program should check the value of 'err'. Any non-zero value indicates an error occurred.

7. Get the data from the file, one record at a time:

```
    ln CALL "GETREC"; reclen, err, bufferl, buffer2
```

where:

'reclen' and 'err' are defined as in step 6.

'bufferl' is a string variable which will contain the data record. It must be "long" enough (have enough space) to contain the longest data record in the file.

'buffer2' is an optional string variable which is only required when the data record contains 256 bytes. Since BASIC string variables have a maximum length of 255 characters. 'bufferl' cannot hold all the data. In this case only, you need to provide a second string variable to contain the extra data. This is the purpose of 'buffer2'.

Upon return from $GETREC, 'bufferl' (and 'buffer2' if used) contains the data record. You can then use BASIC disk I/O statements to write into an RMBASIC data file.

Repeat step 7 until all the records have been read in.

8. CLOSE the data file:

```
    CALL "CLOSE"; err
```

where:

'err' is defined as in step 6.

Error Codes

For the meaning of the error codes, see the Model II "Disk Operating System Manual". Also, you can use the TRSDOS ERROR command to print the meaning of an error code.

For example, if you get the TRSDOS error message for code 15 and you are in BASIC, type:

```
     SY "ERROR 15"
```

TRSDOS will print the meaning of this code:

```
     DISK IS WRITE PROTECTED
```

If you are in TRSDOS, simply type:

```
     ERROR 15
```

TRSDOS Character Codes

Code		Character		
		Key-board	Video Display	
Dec.	Hex.		Scroll mode	Graphics mode
00	00	HOLD		▮
01	01	F1	Turns on blinking cursor	▮
02	02	F2	Turns off cursor	▮
03	03	BREAK * CTRL C		▮
04	04	CTRL D	Turns on steady cursor	▮
05	05	CTRL E		▮
06	06	CTRL F		▮
07	07	CTRL G		▮
08	08	BACKSPACE CTRL H	Backspaces cursor and erases character	▮
09	09	TAB CTRL I	Advance cursor to next 8-character boundary	▮
10	0A	CTRL J	Line feed	▮
11	0B	CTRL K		▮
12	0C	CTRL L		▮
13	0D	ENTER CTRL M	Carriage return	▮
14	0E	CTRL N		▮
15	0F	CTRL O		▯
16	10	CTRL P		▯
17	11	CTRL Q		▯
18	12	CTRL R		▮
19	13	CTRL S		▮
20	14	CTRL T		▮
21	15	CTRL U		▯
22	16	CTRL V		▬
23	17	CTRL W	Erase to end of line	▯
24	18	CTRL X	Erase to end of screen	▯
25	19	CTRL Y	Sets white-on-black mode	▮
26	1A	CTRL Z	Sets black-on-white mode	▮
27	1B	ESC	Clears screen, homes cursor	▮
28	1C	←	Moves cursor back	▯
29	1D	→	Moves cursor forward	▮
30	1E	↑	Sets 80-character mode and clears display	▮
31	1F	↓	Sets 40-character mode and clears display	▯

* BREAK is always intercepted. It will never return a X'03'.

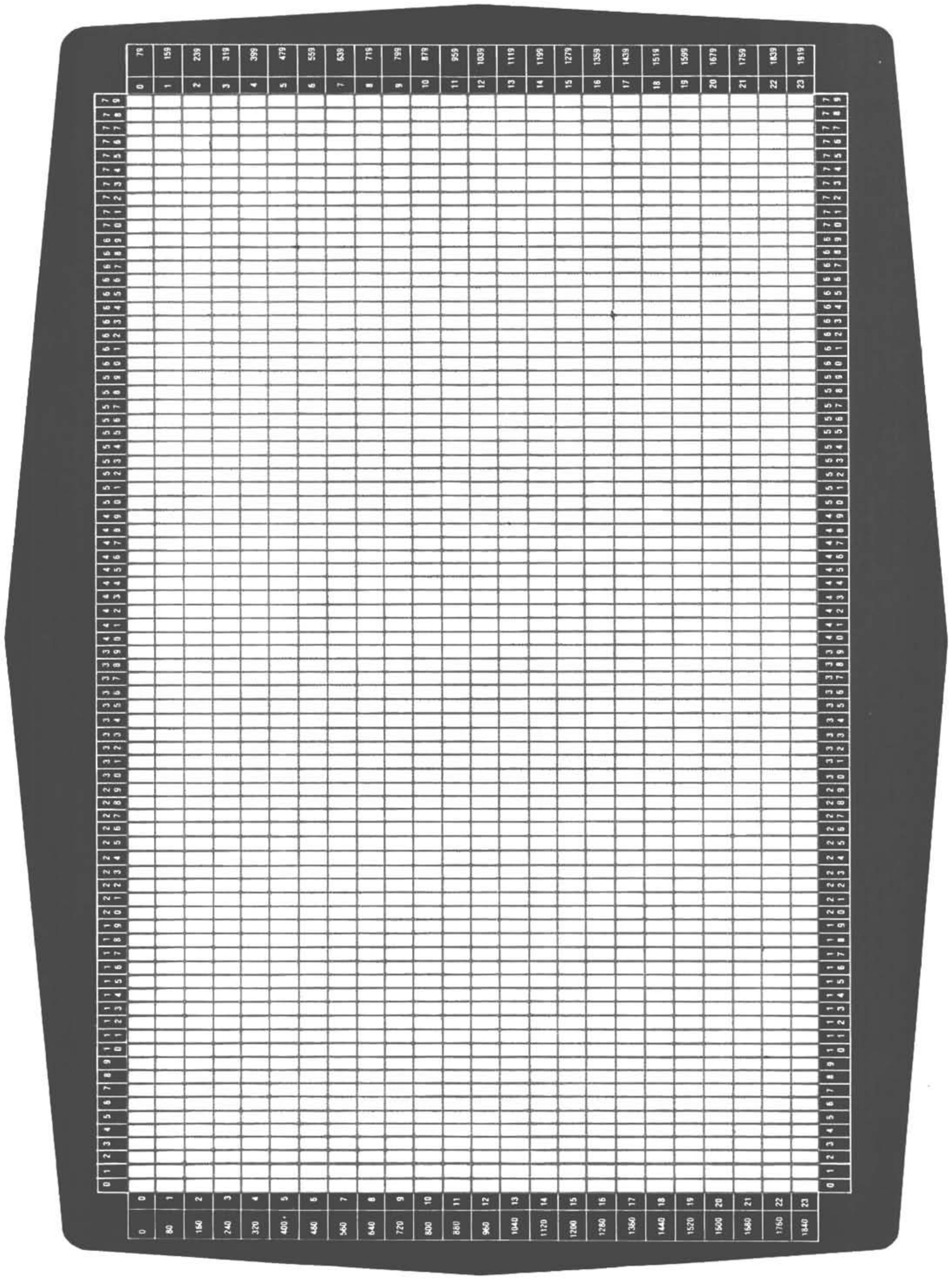
Code		Character		
		Key-board	Video Display	
Dec.	Hex.		Scroll mode	Graphics mode
32	20	Space Bar	b̸	b̸
33	21	!	!	!
34	22	"	"	"
35	23	#	#	#
36	24	$	$	$
37	25	%	%	%
38	26	&	&	&
39	27	'	'	'
40	28	(	(	(
41	29	)	)	)
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	–	–	–
46	2E	.	.	.
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	:
59	3B	;	;	;
60	3C	<	<	<
61	3D	=	=	=
62	3E	>	>	>
63	3F	?	?	?
64	40	@	@	@
65	41	A	A	A
66	42	B	B	B
67	43	C	C	C
68	44	D	D	D

Code		Character		
		Key-board	Video Display	
Dec.	Hex.		Scroll mode	Graphics mode
69	45	E	E	E
70	46	F	F	F
71	47	G	G	G
72	48	H	H	H
73	49	I	I	I
74	4A	J	J	J
75	4B	K	K	K
76	4C	L	L	L
77	4D	M	M	M
78	4E	N	N	N
79	4F	O	O	O
80	50	P	P	P
81	51	Q	Q	Q
82	52	R	R	R
83	53	S	S	S
84	54	T	T	T
85	55	U	U	U
86	56	V	V	V
87	57	W	W	W
88	58	X	X	X
89	59	Y	Y	Y
90	5A	Z	Z	Z
91	5B	[	[	[
92	5C	CTRL 9	\	\
93	5D	]	]	]
94	5E	^	^	^
95	5F	—	—	—
96	60			`
97	61	A	a	a
98	62	B	b	b
99	63	C	c	c
100	64	D	d	d
101	65	E	e	e
102	66	F	f	f
103	67	G	g	g
104	68	H	h	h
105	69	I	i	i

Code		Character		
		Key-	Video Display	
Dec.	Hex.	board	Scroll mode	Graphics mode
106	6A	J	j	j
107	6B	K	k	k
108	6C	L	l	l
109	6D	M	m	m
110	6E	N	m	n
111	6F	O	o	o
112	70	P	p	p
113	71	Q	q	q
114	72	R	r	r
115	73	S	s	s
116	74	T	t	t
117	75	U	u	u
118	76	V	v	v
119	77	W	w	w
120	78	X	x	x
121	79	Y	y	y
122	7A	Z	z	z
123	7B	{	{	{
124	7C	CTRL Ø	¦	¦
125	7D	}	}	}
126	7E	CTRL 6	~	~
127	7F			±
128	80	*		
:	:			
:	:			
248	F8			
249	F9			Sets white-on-black mode
250	FA			Sets black-on-white mode
251	FB			Homes cursor
252	FC			Moves cursor back
253	FD			Moves cursor forward
254	FE			Moves cursor up
255	FF			Moves cursor down

* Codes 128-248 cannot be input from the keyboard or output to the display. When reading the display, a **value** greater than 127 indicates a reverse character corresponding to **value** mod 128.

MODEL II COMPILER BASIC VIDEO DISPLAY WORKSHEET



COMPILER BASIC
OPERATORS AND SPECIAL SIGNS

For information on these operators and special signs, see Chapter 3, "BASIC Concepts".

SPECIAL SIGNS

E	Power of 10
&	Hexadecimal constant

OPERATORS

Numeric

+	Addition
-	Subtraction
*	Multiplication
/	Division
** or ^	Exponentiation
\	Integer Division
MOD	Modulus Arithmetic

String

&	Concatenation

Relational

=	Equals
>< or <>	Not equal to
>= or =>	Greater than or Equal
<= or =<	Less than or Equal
>	Greater than
<	Less than

Logical

AND	Logical AND
OR	Logical OR
NOT	Logical NOT
XOR	Logical XOR

TYPE DECLATATION TAGS

$	String
%	Integer
#	Real

COMPILER BASIC COMMANDS, STATEMENTS, AND FUNCTIONS

WORD	MEANING	PAGE NO
ABS	Append two programs (Function)	6-12
AND	Calculate logical AND (Function)	6-14
APPEND	Append two programs (Command)	2-5
ASC	Get ASCII code (Function)	6-16
ATN	Compute arctangent (Function)	6-18
AUTO	Number lines automatically (Command)	2-7
BREAK	Set or remove program breakpoints (Command)	2-9
CALL	Execute external subroutine (Statement)	6-20
CHAIN	Load and execute next program (Statement)	6-24
CHANGE	Change program lines (Command)	2-10
CHR$	Get character ASCII or control code (Function)	6-25
CLEAR	Clear all programs from memory (Command)	2-12
CLOSE	Close disk file (Statement)	6-27
COM	Allocate common variable area (Statement)	6-28
COMPILE	Compile BASIC program (Command)	2-13
COS	Compute cosine (Function)	6-30
CRT	Position cursor (Function)	6-32
CRTG	Print in graphics mode (Function)	6-35
CRTI$	Read video display (Function)	6-39
CRTR	Move cursor (Function)	6-42
CRTX	Find cursor position (Function)	6-44
CRTY	Find cursor position (Function)	6-44
CVD	Convert Integer to Real (Function)	6-46
CVI	Convert Real to Integer (Function)	6-48
DATA	Store program-data (Statement)	6-50
DATE$	Get today's date (Function)	6-52
DEF	Define function (Statement)	6-54
DELETE	Delete record from disk file (Statement)	6-57
DELETE	Erase program lines from memory (Command)	2-17
DIG	Compute number of numeric characters (Function)	6-58
DIM	Define string variables & arrays (Statement)	6-60
DISPLAY	Display variable contents (Command)	2-18
DUPLICATE	Duplicate program statements (Command)	2-19
END	Terminate program compilation (Statement)	6-65
EOF	Notify if end of file (Function)	6-67
ERR	Get error code (Function)	6-68
ERROR	Simulate error (Statement)	6-69
EXP	Compute natural exponential (Function)	6-70
EXP10	Compute base 10 exponential (Function)	6-71
EXT	Define address of external program (Statement)	6-72
FOR/NEXT	Establish program loop (Statement)	6-73
GO	Start or continue program execution (Command)	2-20
GOSUB	Go to specified subroutine (Statement)	6-76

GOTO	Go to specified line number (Statement)	6-78
HEX$	Compute hexadecimal value (Function)	6-79
HVL	Convert hexadecimal string (Function)	6-81
IF...	Test conditional expression (Statement)	6-83
THEN...		6-83
ELSE		6-83
INKEY$	Get keyboard character if available (Function)	6-86
INPUT	Input data (Statement)	6-87
INPUT from a disk file	Input data from disk file (Statement)	6-92
INPUT USING	Input formatted data (Statement)	6-94
INPUT USING from a disk file	Input formatted data from a disk file (Statement)	6-99
INPUT$	Input a character string (Function)	6-101
INT	Convert to integer value (Function)	6-103
INTEGER	Define variables as integers (Statement)	6-104
KILL	Delete file from disk (Command)	2-21
KILL	Kill disk file (Statement)	6-106
LEN	Get length of string (Function)	6-107
LINE INPUT	Input a line of Data (Statement)	6-108
LINE INPUT from a disk file	Input line from a disk file (Statement)	6-110
LIST	Display program lines (Command)	2-22
LOAD	Load compiled BASIC programs (Command)	2-24
LOG	Compute natural logarithm (Function)	6-112
LOG10	Compute base 10 logarithm (Function)	6-113
LPRINT	Print on line printer (Statement)	6-114
LPRINT USING	Print using format on line printer(Statement)	6-116
MERGE	Merge disk program with resident program (Command)	2-25
NEW	Erase BASIC program from memory (Command)	2-27
OLD	Load BASIC source program (Command)	2-28
ON BREAK GOTO	Enable a <BREAK> handling routine (Statement)	6-118
ON ERROR GOTO	Set up error-trapping routine (Statement)	6-120
ON... GOSUB	Test and branch to subroutine (Statement)	6-122
ON... GOTO	Test and branch to different program line (Statement)	6-123

OPEN	Open disk file (Statement)	6-125
OR	Calculate logical OR (Function)	6-127
POS	Search for specified string (Function)	6-129
PRINT	Print on video display (Statement)	6-131
PRINT to a disk file	Print to disk (Statement)	6-135
PRINT USING	Print using format (Statement)	6-137
PRINT USING to a disk file	Print using format to disk file (Statement)	6-142
RANDONIZE	Reseed random number generator (Statement)	6-144
READ	Get value from DATA Statement (Statement)	6-146
READ from a disk file	Read contents of disk file (Statement)	6-148
REAL	Define variables as real numbers (Statement)	6-150
REM	Comment line (remarks) (Statement)	6-152
RENUMBER	Renumber program (Command)	2-29
RESET BREAK	Disable the <BREAK> handling routine (Statement)	6-153
RESET ERROR	Disable error handling (Statement)	6-154
RESET GOSUB	Clear all returns (Statement)	6-156
RESTORE	Reset data pointer (Statement)	6-158
RESUME	Terminate error trapping routine(Statement)	6-160
RETURN	Return control to calling program(Statement)	6-162
RND	Generate pseudorandom number (Function)	6-163
RUN	Execute program (Command)	2-30
SAVE	Save BASIC source program on disk (Command)	2-31
SEG$	Get substring (Function)	6-165
SGN	Get sign (Function)	6-166
SIN	Compute sine (Function)	6-168
SIZE	Print used and unused memory (Command)	2-33
SQR	Compute square root (Function)	6-170
STEP	Execute portion of program (Command)	2-34
STOP	Stop program execution (Statement)	6-172
STR$	Convert to string representation (Function)	6-173
STRING	Define variables as strings (Statement)	6-176
STRING$	Return string of characters (Function)	6-178
SUB	Name and Define subprogram (Statement)	6-179
SUBEND	End subprogram (Statement)	6-181
SWAP	Exchange values of variables (Statement)	6-182
SYSTEM	Return to TRSDOS (Command)	2-35
SYSTEM	Execute a TRSDOS command (Statement)	6-184
TAB	Tab to position (Function)	6-185
TAN	Compute tangent (Function)	6-186

TIME$	Get the time (Function)	6-188
TRACE ON, TRACE OFF	Turn tracer on, off (Command)	2-36
VAL	Evaluate string (Function)	6-190
WRITE to a disk file	Write to disk (Statement)	6-192
XOR	Calculate exclusive OR (Function)	6-194

INDEX

Radio Shack™

TRS-80

MODEL II
MICRO
COMPUTER

Compiler BASIC Development System

Cat. No. 26-4705