TI-99/4A®

User's Handbook

WEBER SYSTEMS INCORPORATED

TI-99/4A®
User's Handbook

TI-99/4A®
USER'S HANDBOOK

by
WSI Staff

Weber Systems, Inc.
Cleveland, Ohio

TI-99/4A® User's Handbook

Library of Congress Catalog Card Number: 83-060591
ISBN: 0-938862-49-9

Typesetting and Layout: Tina Koran, Jana Butler, and Zemagraphic

CONTENTS

INTRODUCTION

This book is intended to be both a tutorial and an ongoing reference guide to the TI-99/4A computer and its related peripherals. It is a useful guide for beginners and experienced users as well.

The book is divided into three sections. The first section of the book contains an explanation of the techniques used to program the TI-99/4A in TI BASIC and TI Extended BASIC. This section does not assume any prior knowledge of computing, and is written in a concise manner that can be easily read and understood.

The second section of the book is a complete reference guide to all of the TI BASIC and TI Extended BASIC functions, commands and statements. This section is particularly useful for experienced users who are familiar with programming techniques but have difficulty remembering the exact format of each statement.

The final section of the book contains explanations of the usage of TI-99/4A peripheral equipment. An entire chapter is dedicated to each of the most commonly used TI peripherals: the Program Recorder, Disk Drive and Printer.

Several appendices are also included for quick reference. These include error and warning messages, characters codes, as well as several others.

CHAPTER 1.
THE TI-99/4A COMPUTER: INTRODUCTION, INSTALLATION AND OPERATION

INTRODUCTION

In this book, we will describe the TI-99/4A personal computer, as well as many of the various peripherals available for use with the TI-99/4A, such as the TI Peripheral Expansion System, TI 1250 and 1850 Disk Drives, TI Program Recorder, and the TI-99/4 Impact Printer. It should be noted that the TI-99/4A is an improved model of the original Texas Instruments personal computer, the TI-99/4.

Chapter 1 includes a discussion of the features of the TI-99/4A and available peripherals, and provides step by step intructions on the installation of the computer system.

Chapters 2, 3 and 4 provide an explanation of the programming of the TI-99/4A computer. Chapter 2 provides information that allows the user to write programs in the standard version of TI BASIC. Chapter 3 contains an explanation of the TI Extended BASIC programming language. Chapter 4 is a discussion of the techniques used to generate sound and graphics with the TI-99/4A computer.

Chapter 5 is a reference guide to the commands, statements and functions of TI BASIC as well as TI Extended BASIC.

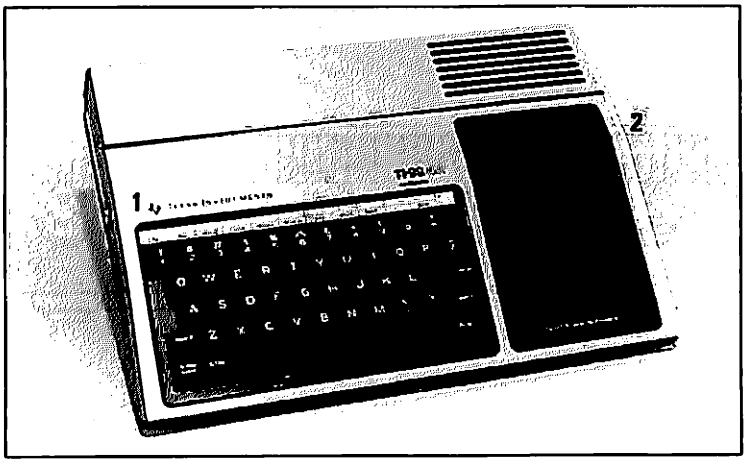
Chapter 6 provides an explanation of the TI Program Recorder. This device allows programs and data to be stored on ordinary cassette tapes. The TI 1250 and 1850 Disk Drives are described in detail in chapter 7. The TI-99/4 Impact Printer is explained in chapter 8.

TI-99/4A OVERVIEW

The TI-99/4A is pictured in Illustrations 1-1, 1-2, and 1-3. Notice the following features of the computer console.

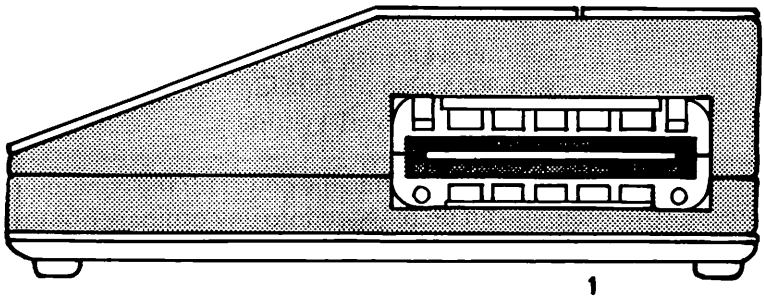
Keyboard --	Used for inputting instructions and information into the TI-99/4A.
Command Module Slot --	Used to accept plug-in TI-99/4A program modules.
Peripheral Expansion Port --	Used to connect external peripherals to the computer console.
Cassette Port --	Used to connect the Program Recorder.
Power Supply Receptacle --	Used to connect the power supply to the TI-99/4A.
Audio/Video Output Jack --	5 pin jack used to connect TV set or monitor.
Wired Remote Port --	Used to connect Wired Remote Controllers (game controllers, etc.)

Illustration 1-1. TI-99/4A (Top View)



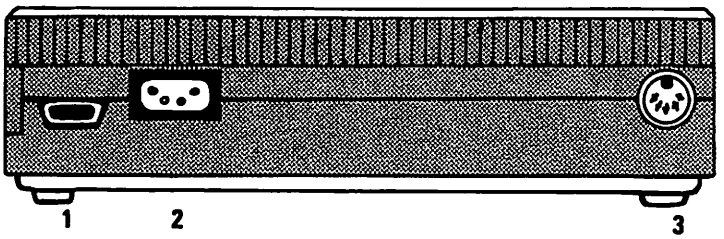
1. Keyboard 2. Command Module Slot.

Illustration 1-2. TI-99/4A (Side View)



1. Peripheral Expansion Port (cover open).

Illustration 1-3. TI-99/4A (Rear View)



1. Cassette Port 2. Power Supply Receptacle 3. Audio/Video Output Jack.

TI-99/4A SPECIFICATIONS

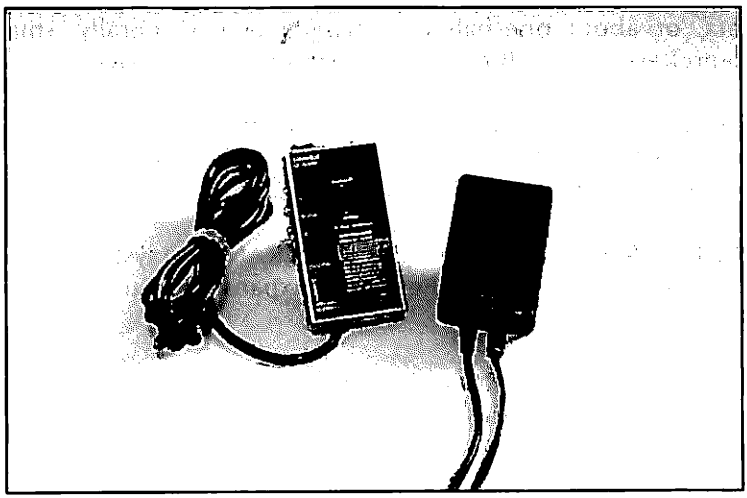
The following components are included with the TI-99/4A computer console.

- AC Power Adapter
- TI-900 Video Modulator
- Keyboard Overlay Strips

The AC Power Adapter provides suitable power for the computer console. The Adapter can be plugged into any ordinary household outlet.

Two types of Power Adapters have been produced for the TI-99/4A computer. One model is plugged directly into an outlet, but the other has a power cord and a plug. Either of these two types of Power Adapters has a small plug that can be connected to the power cord receptacle on the back of the computer console. (see Illustration 1-3).

Illustration 1-4. AC Power Adapter and RF Modulator



The Power Adapter that plugs directly into the wall must be securely in place. A screw is included with this type of adapter to fasten it on the outlet plate.

If a TV set is used for video display, the TI-900 Video Modulator is used to convert the computer's video output to a television signal.

If the Texas Instruments color monitor is used, it is not necessary to use the RF modulator.

Computer Memory

The computer cannot perform any calculations or process any data unless a set of instructions (a program) and data are provided. The part of the computer that is used to store programs and data is called the random access memory (or RAM). The computer can only process data that is stored in RAM. As a result, computers with more random access memory can handle longer programs and more data than computers with less RAM.

A convenient unit of memory capacity is the kilobyte, or simply K. One K is enough memory capacity to store 1024 characters of data, or about one-half of a typed page. Generally, small computers contain 16K, 32K, 48K, 64K or 128K of RAM.

The random access memory cannot be maintained unless the computer is powered on. As a result, the contents of RAM are erased when the power is shut off.

The TI-99/4A computer contains 16K of RAM. The system can be expanded to 48K with the Peripheral Expansion System and the Memory Expansion Card.

Read-only memory (ROM) is similiar in principle to RAM, but has two distinct features. The contents of the read-only memory cannot be changed. Also, the contents of ROM remain intact when the computer is shut off.

The programs that are permanently stored in ROM are used to maintain the operations of the computer.

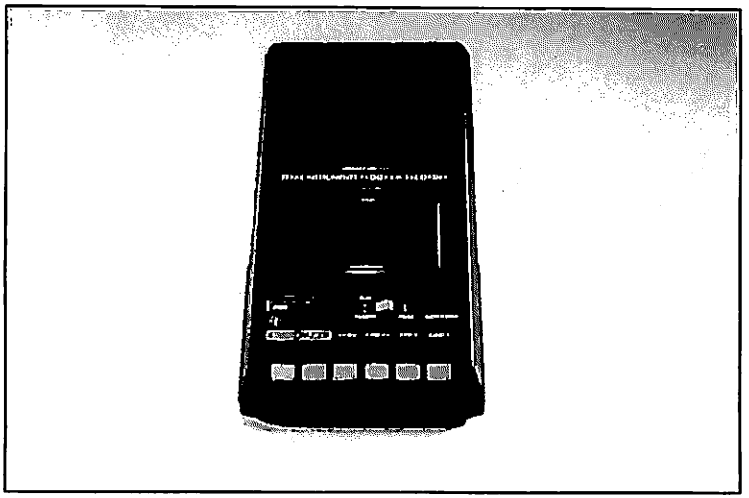
TI-99/4A SYSTEM PERIPHERALS & ACCESSORIES

A complete TI-99/4A system includes the main computer console, monitor, Peripheral Expansion System, printer and Program Recorder.

Program Recorder

The Program Recorder can be used to store programs or data. This allows programs and data to be stored and recovered at a later date. Several thousand characters of information can be stored on an ordinary cassette tape.

Illustration 1-5. Program Recorder

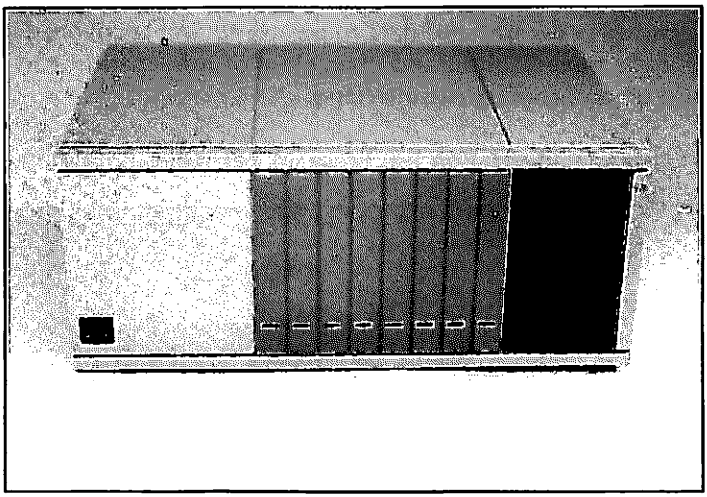


Peripheral Expansion System

The Peripheral Expansion System is the foundation of an expanded TI-99/4A system. It allows the addition of special purpose peripheral devices, such as the Disk Memory System, an RS-232 Interface, and the 32K RAM Memory Expansion.

Each of the peripheral devices require a card to be installed in the Peripheral Expansion box. A card is simply an electronic device that allows the computer to perform a specific function. A total of 8 cards can be installed in the Peripheral Expansion box.

Illustration 1-6. Peripheral Expansion System with TI 1250 Disk Drive



Disk Memory System Card

The Disk Memory System Card contains the electronic circuitry required for the TI-99/4A to use disk drives. With this card mounted in the Peripheral Expansion System, the TI-99/4A can access up to 3 disk drives.

TI 1250 & 1850 Disk Drives

A disk drive is a much more efficient device for storing data than a cassette recorder. A disk drive allows greater storage capacity, quicker access to data, and fewer errors in the transfer of data.

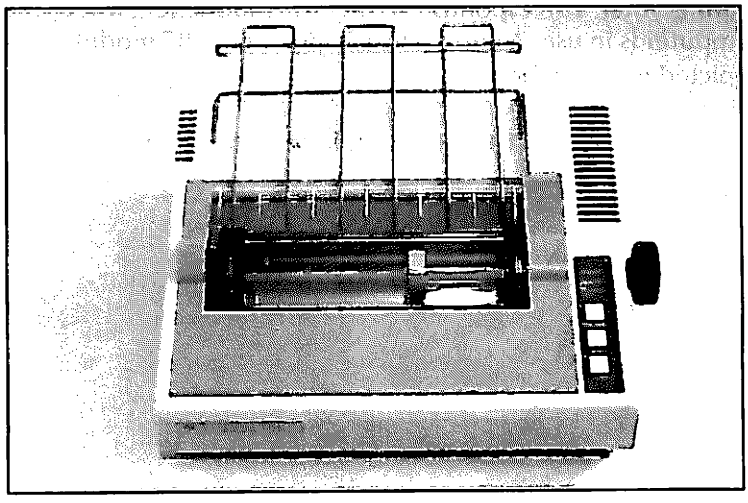
The TI 1250 and 1850 disk drives are designed for use with the TI-99/4A. They are virtually identical except for the fact the 1250 is designed to mount inside the Expansion System cabinet, while the 1850 has its own enclosure. Both the 1250 and the 1850 use single-sided, single density 5 ¼" diskettes, and can store approximately 90,000 characters on each diskette.

TI-99/4 Impact Printer

The TI-99/4 Impact Printer is capable of printing both text and graphics characters. The printer outputs characters at a rate of 80 per second, and uses standard, tractor feed paper.

The Impact Printer must be connected to the computer through an RS-232 Interface.

Illustration 1-7. TI-99/4 Printer



RS-232 Interface Card

The RS-232 Interface Card plugs into the Expansion System, and allows the addition of input or output devices such as a modem or printer.

RS-232 is a code that was developed in order to standardize the exchange of data along telephone lines.

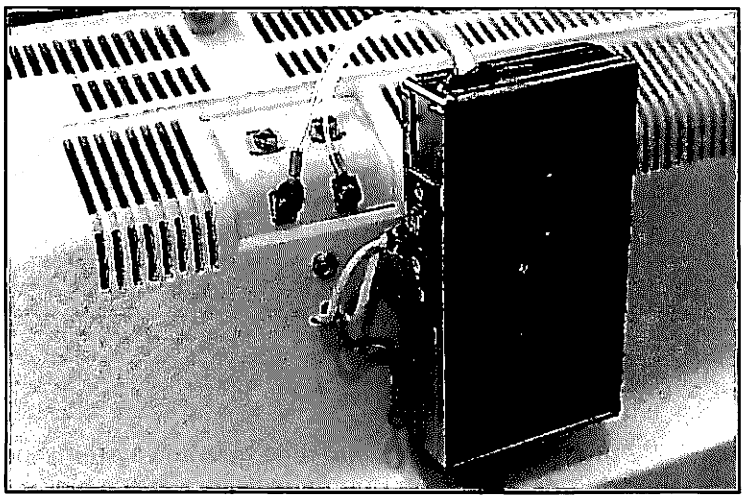
32K RAM Memory Expansion Card

The Memory Expansion Card is plugged into the Expansion System to add 32K of RAM to the original 16K. This extra memory

cannot be used with standard TI BASIC. However, it may be used with TI Extended BASIC, TI LOGO, Editor/Assembler, or several other Solid State Software Command Modules.

The RF modulator contains another switch labeled CHANNEL SELECT. This switch determines the channel on which the television will receive the computer output signal. The switch can be used to select either channel 3 or channel 4. Set the switch to the channel that receives the weaker signal in your area. Be sure that the channel selector on the television is tuned to the channel that corresponds to the RF modulator when the computer is in use. A correct installation of the RF modulator is depicted in Illustration 1-8.

Illustration 1-8. RF Modulator Installation



If a television is not used with the TI-99/4A, the TI color monitor can be used instead. The monitor is supplied with an audio/video output jack on the back of the computer console. The other end of the monitor cable contains two plugs. The plugs correspond to the two jacks on the back of the monitor. The two plugs on the cable are shaped differently to prevent them from being confused.

Solid State Speech Synthesizer

The Solid State Speech Synthesizer can be used to produce electronically simulated speech. The speed synthesizer can produce a limited number of words when it is used in conjunction with Extended BASIC. When the speech synthesizer is used with the Terminal Emulator command module, its vocabulary is unlimited. Unfortunately, the speech synthesizer cannot be used with standard BASIC, but it can be used with many specialized command modules.

INSTALLATION

The installation of the TI-99/4A computer is not at all difficult. It only takes a few minutes to set up the equipment and make the appropriate connections.

It is generally a good idea to save the carton and packing materials that are supplied with the computer. The packaging allows the computer to be safely shipped or stored in the future.

Once the computer has been removed from the carton, choose a convenient location for the equipment. At least two electrical outlets are required for the computer. One outlet is required for the computer console, and another is needed for a television or monitor. Additional outlets may be required for any peripheral equipment.

The computer console should be placed in a location where it will not be exposed to extreme heat or humidity. Also, the console should be protected from direct sunlight, excessive dust, moisture, etc.

The power switch for the TI-99/4A is located on the right hand side of the front of the console. Be sure that the switch is in the off position (left) during the installation procedure.

Begin by plugging the power supply into a wall outlet. If your computer is supplied with a power supply that has a power cord, simply place the power supply in a safe place near the computer console.

If your power supply plugs directly into an outlet, be sure that it is firmly in place. If possible, use the screw that is provided with the power supply to fasten the unit to the outlet plate.

Proceed by connecting the power supply to the computer console. The power supply plug is accommodated by the power supply receptacle on the back of the computer console. Be sure that the four connector pins in the receptacle are lined up with the four holes in the power supply plug.

If the computer console is to be used with a television, an RF modulator must be used. An RF modulator is a device that converts the audio and video output of the computer into a signal that is compatible with a standard television.

The RF modulator supplied with the TI-99/4A has a cord with a round, 5 pin connector. This plug is accommodated by the round receptacle on the back of the computer console.

Once again, be sure that the pins in the plug and the holes in the receptacle are aligned properly. If the plug is forced into the receptacle without being properly aligned, both the plug and receptacle may be damaged.

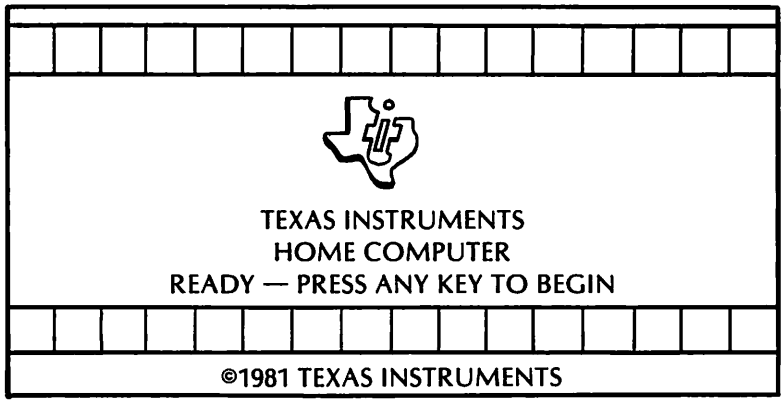
Proceed by removing any antenna leads that may be connected to the VHF terminals on the television. Connect the antenna leads (if any) to the terminals labeled TV ANTENNA IN on the RF modulator.

The RF modulator contains a switch that is labeled MODULATOR/TV ANTENNA. This switch allows the television to operate normally when the computer is not in use. When the switch is in the MODULATOR position, the television receives the signal from the computer. When the switch is in the TV ANTENNA position, the television receives the signal from the antenna.

When the power supply has been connected to the console, and a monitor or television has been installed, the system is ready to be powered on. Begin by turning on the television or monitor. Proceed by moving the computer console power switch to the ON position.

The red power indicator on the front of the computer should be illuminated and the following should appear on the screen.

Illustration 1-9. The Master Title



If the initial display does not appear as in the previous illustration, consult the following table for a list of solutions to common installation problems.

Table 1-1. Troubleshooting Guide

Problem	Possible Cause	Solution
No video display with power light off.	Power Adapter unplugged from outlet.	Check connection between wall outlet and Power Adapter.
	Power Adapter not connected to to computer console.	Check connection between Power Socket and Power Adapter.

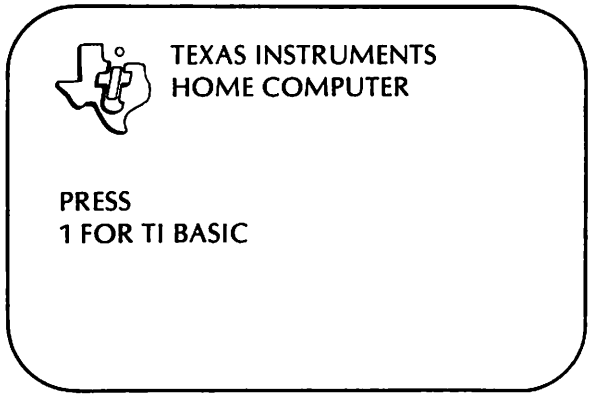
Table 1-1. Troubleshooting Guide

Problem	Possible Cause	Solution
No video display with power light on.	TV tuned to wrong channel.	Be sure that the TV is tuned to the channel indicated on the RF Modulator.
	RF Modulator not connected to console.	Check connection between RF Modulator and console.
	RF Modulator not properly connected to TV.	Be sure that the twin leads are connected to the VHF terminals on the TV-not UHF.
No color or incorrect color on display.	TV is improperly tuned or color is adjusted incorrectly	Tune TV and/or adjust color.

Getting Started

The master title screen allows the colors of the display to be adjusted. When the master title screen is displayed, the display changes to the master selection menu when a character is typed.

The master selection menu is a list of options that appear on the display. If no command modules are plugged into the console, the menu appears as follows.



When a command module is in use, additional selections appear on the menu. Any selection on the menu can be chosen by typing the appropriate number.

When the "1" key is pressed, the display is cleared and the following message will be displayed.

TI BASIC READY

>■

Prompt

When the TI-99/4A is ready to accept data from the keyboard, an angle bracket (>) is displayed at the left edge of the display. This symbol is generally called a prompt.

Throughout this book, the prompt (>) is used to designate program lines or statements that are entered by the user.

Cursor

The small rectangle that appears to the right of the prompt is called the cursor. The cursor is a symbol that is used to indicate the position that the next typed character will occupy.

The cursor is easily recognized because it continually blinks on and off.

Display

The TI-99/4A computer displays 24 lines of output on a television or monitor. Each line can contain up to 28 characters. A line on the display is commonly called a display line.

The computer is not restricted to information that can appear on one line of the display. When the standard version of BASIC is in use, the computer can accept data that is up to 4 lines long. The maximum length of a line of information is called the computer's logical line length.

In the standard version of TI BASIC, the logical line is four display lines long. The computer's logical line is longer when TI Extended BASIC is in use.

TI-99/4A KEYBOARD

The TI-99/4A Keyboard contains many of the same keys arranged in the same order as a standard typewriter. However, the keyboard also contains several additional keys, such as CTRL and FCTN.

Many of the keys have special symbols on the front in addition to the symbols on the top of the keys. Each of these keys are assigned a special function in addition to the regular function of the key.

Any problems that occur as a result of keyboard entry can be solved by turning the computer off and turning it on again. As a result, experimentation with the keyboard cannot damage the computer.

SHIFT

The SHIFT keys on the TI-99/4A are used in the same manner as the SHIFT keys on a typewriter. When one of the SHIFT keys is pressed, each lettered key will generate an upper case character rather than a lower case character.

The TI-99/4 has lowercase characters that resemble the uppercase characters. The computer does not have a unique set of lowercase letters. Lowercase letters are merely a smaller version of the uppercase characters.

Each numbered key generates a unique symbol when a SHIFT key is pressed. The symbols that are represented by these keys are displayed on each key above the corresponding number.

Five keys on the right side of the keyboard are used only to generate symbols. These keys have the following markings.

+ : — < >
= ; / , .

Each of these keys are used to generate the symbol that appears on the lower half of the key. However, when one of the SHIFT keys is pressed, each of these five keys can be used to generate the symbol on the upper half of the key.

ALPHA LOCK

When the ALPHA LOCK key is pressed, it does not return to its original position. Instead, the ALPHA LOCK key remains down when it is pressed. The key does not return to its original position until it is pressed again.

When the ALPHA LOCK key is depressed, the lettered keys on the keyboard will generate the uppercase characters only. The numbered keys and the five symbol keys will not be affected by the ALPHA LOCK key.

ENTER

The ENTER key on the computer is used to enter data and move the cursor to the beginning of the next line on the display.

When data is typed at the keyboard, the information is not actually entered into the computer until the ENTER key is pressed.

The ENTER key is sometimes called the "carriage return key" because its function is similar to the carriage return key on a typewriter.

CTRL

The key that is labeled CTRL is generally called the "control key." This key is not commonly used with the TI-99/4A computer.

The CTRL key is used with other keys on the keyboard to generate special commands. The "control commands" are generally used when the computer is used as a terminal for a larger computer system.

FCTN

The key that is labeled FCTN is called the "function key." This key is used in conjunction with several other keys on the keyboard to perform special functions.

Several keys on the keyboard have special functions when the FCTN key is depressed. These special functions will be described in this book with the following notation.

FCTN X

The character that follows "FCTN" corresponds to the marking on a key. Be sure to press the specified key while the FCTN key is being held down.

A template is provided with the computer to help the user remember the function commands. The labels that appear on the template (DEL, INS, ERASE . . .) correspond to the function commands for the top row of keys on the keyboard.

The FCTN key can also be used to generate the following special characters.

~ [ ] — ? ' " { } | \ '

These characters can be generated by holding down the function key and typing the key that has the corresponding character printed on the front of the key.

Correcting Keyboard Entry Errors

When information is typed incorrectly, there are five commands that can be used to correct the entries before the data has been entered into memory.

These four commands can be used to move the cursor to the left, move the cursor to the right, insert data or delete data.

Cursor Left (FCTN S)

The S key on the keyboard has an arrow on the front of the key that points to the left. When the S key is typed while the function key is held down, the cursor moves one space toward the left edge of the display.

If the cursor is located at the beginning of a logical line, the Cursor Left command has no effect.

When both the function key and the S key are held down, the Cursor Left command will be automatically repeated. This feature allows the cursor to be moved quickly and easily.

Cursor Right (FCTN D)

The D key on the keyboard has an arrow on the front of the key that points to the right. When the D key is typed while the function key is held down, the cursor moves one space toward the right edge of the display. If the cursor is moved past the end of a display line, it automatically moves to the beginning of the next line.

If the cursor is located at the end of a logical line, the Cursor Right command has no effect.

When both the function key and the D key are held down, the Cursor Right command is automatically repeated. This feature allows the cursor to be moved quickly and easily.

INS (FCTN 2)

The INS command is used to insert data. When the insert command is executed, any subsequent typed characters will be displayed at the cursor location. Each character that is inserted causes the cursor to move one space to the right.

As the cursor is moved toward the end of the logical line, the data on the right side of the cursor will also be moved to the right. This allows the data to be inserted in the line without overwriting any of the existing data.

The INS command is automatically repeated when the FCTN and 2 keys are both held down.

DEL (FCTN 1)

The DEL command is used to delete data. When the delete command is executed, the character that occupies the current position of the cursor will be deleted. Each time a character is deleted, the data that appears to the right of the cursor will be moved one space to the left.

When the DEL command is executed, the cursor does not move. The data that appears to the right of the cursor will be deleted one character at a time.

When the FCTN key and the 1 key are both held down, the DEL command will be repeated automatically.

ERASE (FCTN 3)

The ERASE command is used to delete an entire logical line of data. When the ERASE command is executed, the cursor will be returned to the first position on the logical line.

Special Function Keys

QUIT (FCTN =)

The QUIT command is used to reset the computer system. When the QUIT command is executed, the computer's memory will be cleared. As a result, pressing the FCTN and = keys simultaneously causes the program and data in the computer's memory to be erased.

The QUIT command also causes the master title screen to be displayed.

CLEAR (FCTN 4)

The CLEAR command is generally used to stop the program that is currently being executed. The CLEAR command does not eliminate any of the program or data that is currently in use. As a result, the FCTN and 4 keys are used simultaneously to stop the execution of a program.

When the CLEAR command is executed, a message will be displayed that indicates the program line/number that was being executed when the command was issued.

The CONTINUE command can be used to resume the execution of a program that was halted by the CLEAR command.

CHAPTER 2.
PROGRAMMING THE TI-99/4A IN STANDARD BASIC

INTRODUCTION

BASIC is probably the most widely used programming language for microcomputers, with the TI-99/4A being no exception.

Unfortunately, there are many versions of BASIC that are used with various computers. As a result, the version of BASIC that is used with Texas Instruments computers is not the same as the versions of BASIC used with other computers. However, the fundamentals of the BASIC language are the same, regardless of the version.

Immediate & Program Modes

The immediate mode is also known as the direct or the calculator mode. In the immediate mode, most BASIC command entries result in the instructions being executed without delay. For example, if the following immediate mode line was entered,

```
PRINT "JIM SMITH"
```

the following would be displayed on the video screen:

```
JIM SMITH
```

In the program or indirect mode, the computer accepts program lines into memory, where they are stored for later execution. This stored program is executed when the RUN command is entered.

Illustration 2-1 contains an example of the entry of a program in the program mode and its execution.

Illustration 2-1. Program Mode Entry & Execution

```
>10 PRINT "JIM SMITH"
>20 PRINT "1220 EUCLID AVE"
>30 PRINT "CLEVELAND, OH 44122"
>40 END
>RUN
 JIM SMITH
 1220 EUCLID AVE
 CLEVELAND, OH 44122
```

Line Numbers

In the program mode, program lines must begin with a line number. A line number is a number entered at the beginning of a program line. The line number at the beginning of a program line is the only difference between a program line and an immediate mode line.

Line numbers between 1 and 32767 can be used with the TI-99/4A computer.

A program cannot contain two lines that begin with the same line number. If the same line number is used more than once in a program, the most recently entered line will replace the original.

The order in which the statements of a BASIC program are executed is determined by the line numbers. The statement with the lowest line number will be executed first, followed by the program lines with higher line numbers.

Program lines do not need to be entered in the correct sequence. Program lines are executed according to the order of their line numbers, regardless of the order in which they are entered.

Adding program lines to the program stored in the computer's

memory is very easy. Simply type in the line number followed by the appropriate program statement.

The line will be inserted in the program in the position indicated by its line number. For example, by adding the following line to the program in Illustration 2-1,

```
35 PRINT "216-777-5579"
```

the phone number for Jim Smith will be displayed on the line following his city, state, and zip.

Program lines can be deleted by typing the line number of the line to be deleted, followed by Enter. For example, the following entry,

```
30
```

would result in line 30 being deleted.

Program lines can be changed by merely retyping the new line. The existing line in the computer's memory will be replaced with the new line. For example, the following entry,

```
10 PRINT "THOMAS HILL"
```

would result in "THOMAS HILL" being output rather than "JIM SMITH" in the program in Illustration 2-1.

NUMBER

The NUMBER command (or simply NUM) can be used to generate line numbers automatically.

When a NUMBER command is executed, line number 100 is displayed on the screen, and a new line number is generated each time a program line is entered. Each subsequent line number is greater than the previous line number by a value of 10. As a result, the line numbers are generated with the following sequence.

100, 110, 120, 130 ...

If a line number is generated for a line that already exists in the program, the entire line will be displayed along with the line number. If you do not wish to change the program line, simply press the ENTER key to generate the next line number.

The NUMBER command can also be used to generate a different sequence of line numbers. Two arguments can be used with a NUMBER command to indicate the first line number generated, and the increment between line numbers.

For example, the following NUMBER command can be used to generate the line numbers 10,15,20,25, etc.

NUM 10,5

The automatic line number mode can be exited by simply pressing the ENTER key when a new line number is displayed. The CLEAR command (FCTN 4) can also be used to cancel the automatic line number mode.

NEW Command

The NEW command is used to erase an old program from memory before a new one is typed in.

The TI-99/4A can only store one program in its memory at any one time. If you attempt to enter a new program while another program is already stored in the memory, the new program will be merged with the existing program.

END Statement

Notice the last line in the program in Illustration 2-1. That line consists only of the line number plus the BASIC reserved word END.

The END statement identifies the end of a program, and causes the computer to return to the immediate mode. Generally, the END statement should be the last line in a program.

Actually, TI BASIC does not require an END statement. When the program's final statement is executed, the program will end. However, it is good programming practice to end a BASIC program with an END statement.

Executing a Program

The program in the computer's memory is executed when the RUN command is entered. This is shown in Illustration 2-1. Each time the RUN command is repeated, the program is reexecuted.

Listing a Program

The LIST command can be used to display the program lines that are currently stored in the computer's memory.

When the LIST command is executed, the program in the computer's memory is displayed on the screen. Each line of the program appears initially at the bottom of the display. In order for each subsequent line of the program to appear on the last line of the display, each line of the display must be moved one line toward the top of the display.

As a result, if a program occupies more than 24 display lines, the first lines of the program will be moved off the top of the display in order to accommodate the last lines. This process is called **scrolling.**

The LIST command may also be used to display portions of a program.

If a LIST command includes a line number, only the program line with the specified line number is displayed.

For example, the following command can be used to display the program line that has the line number 100.

```
>LIST 100
```

A LIST command can also be used to display the program lines that have line numbers within a specified range. For example, the following command causes all the program lines that have line numbers between 100 and 200 to be displayed.

```
>LIST 100-200
```

When the LIST command is used with only one line number, a dash can be included either before or after that line number. A dash before the line number causes all the program lines that have a line number less than or equal to the specified value to be displayed. If the dash follows the specified line number, all the program lines from the specified line to the end of the program will be displayed.

For example, the following command causes line 1000 to be displayed along with all the program lines with line numbers greater than 1000.

```
>LIST 1000-
```

Error and Warning Messages

When a statement with an incorrect format has been entered, an error message will be displayed. The error message describes the type of problem that occurred.

If a problem develops while a program is being executed, an error message will also be displayed. An error that occurs during the execution of a program generates an error message that includes a description of the problem as well as the line number of the statement that caused the problem.

When an error occurs in a program, an error message will be displayed and the execution of the program will stop. If a problem occurs in a program that is not serious enough to stop the execution, a warning message will be displayed. Warning messages describe the nature of the problem as well as the line number where the problem occurred.

Appendix A provides explanations of the TI BASIC error and warning messages.

BASIC Data Types

There are two general types of data that can be manipulated by the computer: string and numeric.

Numeric data consists of numeric values and string data consists of numbers, letters, and special characters.

Strings

A **string** consists of one or more characters enclosed within double quotation marks. The following are examples of strings:

"F. SCOTT FITZGERALD"
"149 LEXINGTON AVE"
"NEW YORK, NY 10017"
"212-349-9879"

Notice that a string can contain both letters, numbers, and symbols. Any string containing numbers cannot be used in mathematical operations.

NUMERIC DATA

Floating Decimal Point

Floating decimal point is the standard method of representing numeric data. With floating decimal point numbers, a decimal point is always assumed. Any number of digits can be placed on either side of the decimal point. Even with numbers with no decimal position, a decimal point always is assumed following the number's last digit.

Floating point numbers are displayed with 10 digits of accuracy. For example, the following entry of a 10 digit floating point number,

```
PRINT .5666666666
```

would generate a ten digit display. If an 11 digit floating point number was entered,

PRINT .56666666666

the last digit would not be displayed and the number would be rounded as follows.

.5666666667

Commas may not be included within numeric data. For example, 109000 would be a valid number, while 109,000 would be invalid.

Floating point numbers include integers as well as numbers with decimal positions. The following are examples of floating point numbers.

-.0789
5
77.39
0
+.000001
67.98

Negative floating point numbers should be preceded by a minus sign (-). Positive floating numbers can optionally be preceded with the plus sign (+), however, a floating point number is assumed positive if it doesn't have a sign.

Scientific Notation

TI BASIC uses **scientific notation** to express either extremely large or extremely small numbers. A number in scientific notation has the following format:

±x E ± yy

Where;

± is an optional plus or minus sign.

x is a floating point number. This position of the number is known as the coefficient or mantissa.

E stands for exponent.

yy is a one or two digit exponent. The exponent gives the number of places that the decimal point must be moved to give its true location. The decimal point is moved to the right with the positive exponents. The decimal point is moved to the left with negative exponents.

The following examples specify a number in both standard floating point and scientific notation:

1000000 → 1 E6
.000001 → 1 E-6
57500000 → 5.75 E+07
-.00000479 → -4.79 E-06

Any integers containing 11 or more digits will be expressed in scientific notation as shown in the following example.

```
PRINT 12345678901
1.23457 E+10
```

Notice that the decimal portion of the preceding example contains 6 digits of precision. Any additional digits are rounded off.

Numbers cannot be displayed with an exponent greater than 99. Numbers that have exponents greater than 99 are displayed with two asterisks in the position of the exponent. This indicates that a number was generated that cannot be represented with a two digit exponent.

```
2.47685E+**
```

Numbers that require an exponent that is less than -99 are automatically converted to zero.

The computer cannot manipulate numbers that have an exponent greater than 127. As a result, the following warning message will be displayed when a number is generated with an exponent that is too large.

```
*WARNING
 NUMBER TOO BIG
```

A line number will be included with the warning message if an illegal value was generated during a program.

VARIABLES

So far, we have only discussed data constants. A constant can be defined as a fixed value. The following are examples of string and numeric constants.

```
"JACK NOVET"
"375"
27.59
0
100000
```

A name can be used to express data as well as a constant. Variables are used to express data as a name.

BASIC Variables

A variable can be defined as a name that can represent any one of a group of values. Variables are represented by variable names. These consist of a letter followed optionally by additional letters and/or numbers. The value assumed by a variable is subject to change, depending upon the program statement being executed. The following example program uses the variables A and B.

```
100 LET A = 5.0
200 LET B = 7.0
300 LET A = A + B
```

The variable A is initially assigned a value of 5.0 and B is assigned a value of 7.0. In line 300, the variable A is assigned a new value equal to the sum of variables A and B, which is 12.0. The previous value of A is erased.

Note the use of the LET statement in the preceding example. The LET statement is used to assign a value to a variable. Whenever a LET statement is used in a program, the value of the variable on the left side of the equation is to be replaced with the value appearing on the right.

The reserved word, LET need not actually be included in a LET statement. Both of the following statements have the same effect.

```
100 LET A = 5
200 A = 5
```

BASIC Variable Names

TI BASIC allows any group of up to 15 characters to be used as a variable name--as long as the first character of the group is a capital letter of the alphabet, and as long as the variable name does not duplicate a reserved word (see Appendix D). Examples of reserved words are:

LET, GOTO, IF, READ, DATA

The following are examples of valid BASIC variable names:

A	PRICE
AMOUNT	A7
VALUE	B2A

The following are examples of invalid variable names.

2BB7	END
1A	FOR
PRINT	COS

All of the preceding examples of valid variable names should be used to represent numeric data. Variable names can also be used to represent string data. These are known as **string variables.** String variable names consist of a valid variable name followed by the dollar sign ($). The following are examples of valid string variable names.

A$	NED$
ZIP$	MON$
A7$	N222$

Tables & Arrays

A variable is designed to hold a single data item--either string or numeric. However, some programs require that hundreds or even thousands of variable names be used.

Obviously, the use of thousands of individual variable names could prove extremely cumbersome. To overcome this problem, BASIC allows the use of **subscripted variables.** Subscripted variables are identified with a **subscript.** A subscript is a number appearing within parentheses immediately after the variable name. An example of a group of subscripted variables is given below:

A(0), A(1), A(2), A(3), A(4),..., A(100)

Note that each subscripted variable is a unique variable. In other words, A(0) differs from A(1), A(2), A(3), A(4), etc.

Subscripted variables should be visualized as an array (or table). In our previous example, the data contained in the array defined by A would consist of one row with 101 columns in it. Such an array is a single-dimension array.

TI BASIC allows one, two or three dimensionally subscripted variables. Illustration 2-2 provides a visualization of a two dimensional array of values.

Illustration 2-2. Two-Dimensional Array

Rows \ Columns	0	1	2
0	27.5	29.4	26.4
1	37.8	36.1	35.8
2	26.4	29.5	25.9
3	40.1	35.8	34.9

If the variable name of the example array was X, the value in the upper left corner of the table would be the value of the variable X(0,0). Each element of the table must be specified by the variable name, followed immediately by two subscripts. The first subscript indicates the row number of the specified value. The second subscript indicates the column. The subscripts must be enclosed in parentheses and separated by commas.

OPTION BASE and DIM statements have an affect on the use of subscripted variables. A DIM statement is used to specify the highest subscripts that can be used in an array. An OPTION BASE statement determines the lowest subscript that can be used in an array.

The following DIM statement can be used to reserve an area in the computer's memory for an array with 27 as the highest row number and 14 as the highest column number.

```
DIM Z(27,14)
```

The size an array is only limited by the amount of available memory. If an insufficient amount of memory is available, a MEMORY FULL error will occur when a DIM statement is executed.

A single DIM statement can be used to set the maximum subscript values for more than one array. The specifications of each array must be separated by commas, as demonstrated in the

following statement.

```
100 DIM A(5,3),B(100),C(2,3)
```

Be sure to include a DIM statement in a program before the subscripted variables are referenced. A BAD SUBSCRIPT error will occur if dimensioned variables are used improperly.

DIM statements are required for arrays that have subscripts greater than 10. However, subscripted variables that do not have subscripts greater than 10 do not require DIM statements.

In general, the lowest subscript that can be used in an array is 0. However, an OPTION BASE statement can be used to change the lowest allowable subscript to 1. Although DIM statements can be used to set subscript limits for each array individually, a single OPTION BASE statement affects every array.

The only two values that can be used in an OPTION BASE statement are 0 and 1. As a result, the following two statements are the only possible OPTION BASE statements.

```
OPTION BASE 0
OPTION BASE 1
```

Expressions and Operators

The values of variables and constants are combined to form a new value through the use of **expressions**. An expression is a group of constants, variables and operators that are used to calculate a single value. The value that is calculated reflects the information contained in the expression. A typical expression would appear as follows.

```
27+A8-X
```

TI BASIC allows the use of arithmetic, relational and string operators. Arithmetic operators are used to evaluate expressions based on the values of the numbers in the expression. Relational operators are used to evaluate expressions based on the relative

values of the items in an expression. String operators are used to evaluate expressions that contain string values.

Arithmetic Operators

The symbols used for addition, subtraction, multiplication, division, and exponentiation are known as **arithmetic operators.** In BASIC, the symbols + and - are used for addition and subtraction respectively. The asterisk (*) is used to indicate multiplication, while the slash (/) is used to indicate division.

When a + or – sign precedes a number, the symbol is used to specify that number's sign.

However, when a minus sign is used to change the sign of a value, the operator is called a unary minus. The following statement uses the unary minus operation to change the sign of the variable A.

```
100 LET A = -A
```

Exponentiation is the process of raising a number to a specified power. For example,

$$A^5$$

is equivalent to the following expression:

$$A*A*A*A*A$$

In TI BASIC, exponentiation is indicated by the caret (^). The value to the left of the caret is the base, and the number to the right of the caret is the exponent. As a result, the expression (A^5) could be represented in BASIC as A^5.

The following examples demonstrate the use of the arithmetic operators.

```
37/A + 4^B
50*4-A
23.25/4 + C
```

String Operators

The only string operation used in TI BASIC is string concatenation. This operation is the combining of two strings to form a third string. The result of a concatenation is a combination of the two strings. The ampersand (&) is used to represent this operation.

For example, if the variable A$ is assigned the value "TOOTH" and B$ is assigned the value "ACHE" the strings can be concatenated to form the string "TOOTHACHE". The expression A$ & B$ can be used to concatenate the two strings.

Relational Operators

Relational operators are used to evaluate relational expressions. These types of expressions can be either true or false. The following relational expressions are all true.

```
5>3
3=3
4<7
```

The conditions of true and false are represented by the values -1 and 0 respectively. As a result, a relational expression is always evaluated to one of the two values (0 or -1).

The relational operators are summarized as follows.

```
<  ---> less than
<= ---> less than or equal to
>  ---> greater than
>= ---> greater than or equal to
=  ---> equal to
<> ---> not equal
```

For example, a "less than" expression is equal to -1 only if the value on the left side of the operator is less than the value on the right. Otherwise the expression is equal to zero.

When relational expressions include string values, the string values are compared according to each character in the string.

One character is considered "greater than" another if its ASCII code is a greater number.

The ASCII code is a widely used system that uses numbers to refer to the characters instead of the characters themselves. The ASCII values that correspond to the TI-99/4A character set can be found in Appendix C.

For example, the ASCII code for an upper case A is 65. The code for an upper case B is 66. As a result, the following relational expressions would be considered true.

```
"B" >= "A"
"B" > "A"
"A" <= "B"
"A" < "B"
"A" <> "B"
```

Strings are compared by the ASCII code for each character, one at a time. If the first characters in each of the strings are the same, the second characters in the strings will be compared.

For example, consider the two string values "JOSEPH" and "JOAN". In a relational expression, the first characters of the strings will be compared first. Since both strings begin with "J", the second charactes will be compared next. Since the second characters are also the same, the comparison continues with the third character.

Since the ASCII code for "A" (65) is less than the ASCII code for "S" (83), "JOAN" is considered less than "JOSEPH".

If the end of a string is encountered during a string comparison, the string with the fewer number of characters will be considered less than the longer string. For example, "ABC" would be considered less than "ABCD".

The relational operators can be used to indicate the relative location of strings in alphabetical order.

The following examples demonstrate the use of relational operators with string values. All of the following expressions are true.

"ABC" = "ABC"
"AAB" > "AAA"
"ALFRED" < "ALFREDO"
A$<Z$ where A$ = "ALFRED" and Z$ = "ALFREDO"

Note that all string constants must be enclosed in quotation marks.

Compound Expressions and Order of Evaluation

Most of the preceding examples were **simple expressions.** A simple expression is one which contains just one operator and one or two operands. Simple expressions can be combined to form **compound expressions.** The following are examples of compound expressions.

(A+B)*7-4
(A+B)/(C+D)
(A>B)>(C< >D)

Without a standard means of evaluating compound expressions, the expression 2+3*5 could be evaluated as either 17 or 25 (depending on the order of operations).

As a result, a standard order of operations is used to prevent confusion. Table 2-1 outlines the standard order of evaluation.

Parentheses are used to enclose expressions that should be evaluated before any other operations are performed.

More than one set of parentheses can appear in an expression. If a set of parentheses are located within another set, the expression within the inner set of parentheses will be evaluated first.

If multiple sets of parentheses are not contained within each other, the expression enclosed in the leftmost set of parentheses will be evaluated first.

Table 2-1. Order of Evaluation

	Operator	Description	Priority
Parentheses	( )	Used to alter order of evaluation.	1
Arithmetic Operators	^	Exponentiation	2
	-	Unary Minus	3
	* /	Multiplication Division	4
	+ -	Addition Subtraction	5
Relational Operators	= <>| < > <= >=	Equal To Not Equal To Less Than Greater Than Less Than or Equal To Greater Than or Equal To	6

When the expressions in parentheses have been evaluated, the remaining operations will be performed. The operations will be performed according to the priority levels of Tabel 2-1. The operations in an expression that have the same priority level will be evaluated from left to right.

Mixed Expressions

Since the relational operators return a numeric value (0 or -1), the result of a relational expression can be used within an algebraic expression.

The following examples demonstrate the format of mixed expressions.

```
4*(B>6)
5+(6<>A)
```

The value returned by a relational expression can be used in the same manner as any other value.

For example, the expression 7*(4>3) would be evaluated as -7. The expression in parentheses is equal to -1 because the relation is true. When the result of the relational expression is multiplied by 7, the result is -7.

TI BASIC Statements

The next several sections contain explanations of the most commonly used BASIC statements.

Remark Statements

Remark statements are used to include comments in a program. It is good programming practice to include a liberal number of Remark statements in a program. Comments should be used to describe the logic that is used in a program.

Remark statements must include the keyword REM. This reserved word causes the statement to be ignored by the computer. Even though Remark statements have no effect on the operation of the computer, the comments will be displayed each time the program is listed.

The following statement demonstrates the format of a typical Remark statement.

```
100  REM INPUT ROUTINE
```

Assignment Statements

Assignment statements are used to assign values to variables. The following are examples of assignment statements.

```
100 LET A = 7
200 B = 42
300 NAME$ = "PHIL"
```

Notice that the keyword LET is optional. Generally, LET is

assumed. Both string and numeric variables can be assigned values with an assignment statement.

DATA and READ Statements

Assigning values to a large number of variables with individual assignment statements could prove very cumbersome. DATA and READ statements can be used to assign values to a large number of variables. The following is an example of DATA and READ statements.

```
100 DATA 100, 500, 1000, "JACK"
200 READ A, B, C, D$
```

The DATA statement creates a list of constant values known as a DATA list. The items in the DATA list are assigned sequentially to the variables in the READ statement. A DATA list is depicted in Illustration 2-3.

Illustration 2-3. DATA and READ Statements

Example Statements
100 DATA 10,20,30
200 DATA JUNE,JULY,AUGUST
300 READ A,B,C
400 READ A$,B$,C$

Data List	Corresponding Variables
10	A
20	B
30	C
JUNE	A$
JULY	B$
AUGUST	C$

DATA statements may contain numeric or string values. These values must be separated or **delimited** with commas. DATA

statements may appear at any point in the program.

The DATA list uses a pointer to indicate which value within the list is to be assigned to the next variable in a READ statement. Before the first READ statement is encountered, the DATA list pointer will point at the first value in the DATA list. As values from the DATA list are assigned to variables in the READ statement, the pointer will move sequentially to each successive item in the DATA list.

The values from the DATA list must match the type of variable to which they are assigned in the READ statement. In other words, a string value cannot be assigned to a numeric variable, or vice versa.

A RESTORE statement can be used to move the data pointer to a specific location in the data list. If a RESTORE statement is not used to move the data pointer, each data item in a program can only be read once. The first READ statement will assign the first value in the first DATA statement to the specified variable.

Each data item in the program will be read sequentially beginning with the first data item. When a RESTORE statement is executed, the data pointer can be moved to the first value in any of the program's DATA statements. The particular DATA statement will be specified by the line number argument in the RESTORE statement. When a RESTORE statement does not include a line number, the data pointer will be moved to the first data item in the program. The following example demonstrates the use of a RESTORE statement.

```
100 READ A,B
200 RESTORE
300 READ C,D,E,F
400 RESTORE 700
500 READ G,H
600 DATA 10,20
700 DATA 30,40
```

At line 100, the variables A and B were assigned the values 10 and

20. The DATA statement at line 600 provided the values for the first two variables in the READ Statement.

The RESTORE statement at line 200 returned the data pointer to the beginning of the data. As a result, the values 10 and 20 were once again assigned to variables via a READ statement.

The READ statement at line 300 assigned the values 10,20,30 and 40 to the variables C,D,E and F.

The RESTORE statement at line 400 caused the data pointer to return to the first data item in line 700. As a result, the READ statement at line 500 assigned the values 30 and 40 to the variables G and H.

Outputting Data

In some of the preceding examples, the PRINT statement was used to display data. The PRINT statement can be used to display both numeric and string data.

The following program statement,

```
100 PRINT "VENDOR LIST"
```

would cause the following output when executed.

```
VENDOR LIST
```

The first item in a PRINT statement is displayed at the cursor's current location.

Two data items can be displayed on the same line with a single PRINT statement by separating the string constants or variables in the PRINT statement with commas. The following statements,

```
100 LET A$ = "JOHN"
200 PRINT A$, "BILL", "PETER"
```

would result in the display shown below:

```
JOHN          BILL
PETER
```

The display screen is divided into two separate zones when data is being output to the screen via the PRINT statement. The first zone begins at the far left side of the screen in the first column. The second zone begins in the middle of the display line.

When a comma appears in a PRINT statement, the computer is instructed to print the next value at the beginning of the next print zone. In the preceding example, "JOHN" is printed in print zone 1. The third parameter, "PETER" is displayed in the first zone of the subsequent display line.

If the first value in a PRINT statement consists of more than 13 characters, the next item will be printed at the first print position on the following line. The following example illustrates this principle.

```
>100 LET A$ = "TORONTO,ONTARIO"
>200 PRINT A$, "BILL", "PETER"
>RUN
 TORONTO,ONTARIO
 BILL                PETER
```

A semicolon can also be used to separate the items in a PRINT statement. A semicolon causes the next item in the PRINT statement to be displayed immediately after the preceding item. When semicolons are used to separate items, no blank spaces appear between string values. Numeric values are separated by two spaces.

When a PRINT statement has been executed, the cursor will be moved to the left margin of the following line. This is known as a **carriage return/line feed.**

If a comma or semicolon occurs at the end of a PRINT statement, the carriage return/line feed will be suppressed. If a comma is placed at the end of a PRINT statement, the next PRINT

statement will begin output at the next print zone after the last item is displayed. If a semicolon is placed at the end of a PRINT statement, the next PRINT statement will begin output immediately following the last item displayed.

A DISPLAY statement can also be used to output data to the screen. DISPLAY statements use the same format as PRINT statements, but DISPLAY statements cannot be used to output data to any device other than the video screen.

DISPLAY statements can be used to output string or numeric values. The values in a DISPLAY statement must be separated by commas or semicolons. A comma causes the next output to appear in the next available column. When a semicolon is used to separate values, the values will be output adjacent to each other.

Numeric values that are output with DISPLAY statements will always be preceded by one blank space.

```
>100 A$ = "THOMAS HILL"
>200 DISPLAY "NAME: ";A$
>300 B = 34
>400 DISPLAY "AGE:";B
>RUN
  NAME: THOMAS HILL
  AGE: 34
```

INPUT Statements

An INPUT statement allows values to be assigned to variables while a program is being executed.

When an INPUT statement is executed, the execution of the program cannot continue until data has been entered via the keyboard.

The following statement demonstrates the format of a simple INPUT statement.

```
100 INPUT A
```

When the preceding statement is executed, a tone will sound and a question mark will be displayed to indicate that data is required for the variable A. A keyboard entry should be used to assign a value to the variable. When the ENTER key is pressed, the execution of the program will resume.

The type of data entered at the keyboard must correspond to the type of variable specified in the INPUT statement. If the two types do not correspond, the following warning message will be displayed.

```
*WARNING:
 INPUT ERROR IN 100
TRY AGAIN:
```

When an input error occurs, the INPUT statement will automatically be repeated. As a result, an INPUT statement will not allow the execution of the program to continue until an appropriate value has been entered.

The values for several variables can be assigned with a single INPUT statement. The following example demonstrates the use of an INPUT statement with several variables.

```
200 INPUT A$,B,C$
```

The variables in an INPUT statement must be separated by commas. Also, the data values entered in response to the INPUT statement must be separated by commas.

When an INPUT statement requires values for more than one variable, the values should be entered in the correct sequence and should be separated with commas. Press the ENTER key to end the entry of values.

The types of values in the response must correspond to the types of variables in the INPUT statement. A numeric value will be considered a string if it is entered as a value for a string variable. If the number of data items entered in response to an INPUT

statement did not correspond to the number of variables in the statement, the following warning message would be displayed.

```
*WARNING:
 INPUT ERROR IN 200
TRY AGAIN:
```

String values that contain commas must be enclosed in quotation marks when entered in response to an INPUT statement. However, the quotation marks will not be considered a part of the string value. The quotation marks are required because commas are generally used to separate data items. If a single string value includes a comma, that value would be interpreted by INPUT as two separate values. The following example contains a program that uses INPUT statements to assign string values to variables.

```
>100 INPUT A$,B$,C$
>200 PRINT A$
>300 PRINT B$
>400 PRINT C$
>RUN
  ? DESK, CHAIR, "PENCILS, PENS" ← user's response
  DESK
  CHAIR
  PENCILS, PENS
```

The data PENCILS, PENS would generally be considered two separate values for an INPUT statement. However, the quotation marks allow the data to be considered as a single string value.

It is a good programming practice to include a prompt message in an INPUT statement. A prompt is used to indicate the exact type of data entry that is required. For example, the statement:

```
100 INPUT "CUSTOMER NAME?":A$
```

would cause the following message to be displayed when the INPUT statement was executed.

CUSTOMER NAME?

The prompt message in an INPUT statement must be enclosed in quotation marks and followed by a colon. The list of variables must follow the prompt message.

When a prompt message is included in an INPUT statement, the question mark (?) will not automatically be displayed. If a prompt message is included, the prompt will be displayed exactly as it appears in the INPUT statement.

Loops

A section of a program that is to be repeated more than once is called a **loop**. Loops are used extensively in computer programming to perform calculations for large sets of data.

One of most commonly used loops is the FOR, NEXT loop. The following program example illustrates a typical application of a FOR, NEXT loop.

```
100 FOR J = 1 TO 20
200 X = J^2
300 PRINT X
400 NEXT J
500 END
```

The section of the program from line number 100 to line number 400 is a FOR, NEXT loop. The variable J is referred to as a counter. The statements within the loop (between the FOR and NEXT statements) will be repeated for various values of the counter (J).

The FOR statement contains the upper and lower limits of the counter. The keyword TO is preceded by the initial value of the counter (1), and followed by the final value (20).

Each time the NEXT J statement is executed, the value of the counter will be increased and the loop will be repeated. As a

result, the statements within the loop (lines 200 and 300) will be repeated 20 times. Each time the loop is repeated, the variable J will represent a different value. As a result, the preceding example program will compute the squares of the numbers from 1 to 20.

When the counter (J) is assigned the final value (20), the statements at lines 200 and 300 will be executed for the last time. When the loop has been completed, the program will proceed with the statement following the NEXT J statement.

Any numeric variable can be used as the counter, and any number of program lines can be included within a loop.

In the preceding example, the counter was incremented by one each time the loop was repeated. If it is necessary to increase the counter by a value other than 1, a STEP statement can be included in the FOR statement. The value following the keyword STEP will determine the amount that the counter is to be increased each time the loop is repeated.

The following program contains a FOR, NEXT loop with a counter that will increase by 3 each time the loop is repeated

```
100 FOR K = 1 TO 20 STEP 3
200 X = K^2
300 PRINT X
400 NEXT K
500 END
```

The FOR, NEXT loop in the preceding example will be repreated 7 times. The counter (K) will be assigned the value 1 when the loop is begun. Each time the loop is executed, the value of the counter will be increased by 3. The value of the counter will be 19 during the last repetition of the loop.

FOR, NEXT loops will be repeated until the value of the counter is larger than the final value. The example loop will be executed for the last time when K has been set equal to 19 because one more repetition would cause the value of K to be set to 22. Since

22 is larger than the final value (20), the loop will not be executed once the variable K has been assigned the value 22.

STEP statements can also include negative or fractional values. The correct format for STEP statements is demonstrated in the following examples.

```
FOR J = 10 TO 0 STEP-1
FOR T = 0 TO 100 STEP.25
FOR K = 25 TO 0 STEP-.5
```

Nested Loops

One loop can be placed inside another loop. The innermost loop is known as a **nested** loop. The following program contains a nested loop.

```
100 FOR J = 0 TO 2
200 FOR K = 0 TO 3
300 READ R(J,K)        inner loop    outer loop
400 NEXT K
500 NEXT J
600 DATA 10,11,12,13,14,15
700 DATA 16,17,18,19,20,21
```

The preceding example is used to read data into the numeric array R.

When using nested loops, be sure to end the inner loop before ending the outer loop. In other words, be sure that the entire inner loop is contained within the outer loop.

A program must contain a NEXT statement that corresponds to each FOR statement. If the FOR and NEXT statements are not properly arranged in a program, a FOR-NEXT ERROR will occur.

If a program contains a loop that is improperly nested, a CAN'T DO THAT error will occur.

Conditional Branches

A **branch** is an interruption in the order in which statements are executed in a program. A branch causes a specified program line to be executed regardless of that line's location within the program.

For example, if line number 500 in a program causes a branch to line 1000, the statements in the program with line numbers between 500 and 1000 will be ignored. The program execution will resume at line number 1000 and continue with the subsequent statements.

Conditional branches are statements that are used to branch a program if a specified condition is true.

A conditional branch statement has the following configuration.

IF *expression* THEN *linenumber*

The expression in an IF,THEN statement is used to control the execution of a program. Relational expressions are most commonly used as the control expression in an IF,THEN statement.

If the relational expression is true, the program will execute the statement at the specified line number. The following statement is an example of a conditional branch statement.

```
500 IF A$ = "STOP" THEN 1000
```

The control expression in this statement is A$ = "STOP". When this expression is true, the statement at line number 1000 will be executed. When the program control is branched to line 1000, the execution of the program will continue with the statements that follow line number 1000.

If the control expression is false, the execution of the program will continue with the statement that follows the IF,THEN statement. In other words, the normal execution of the program

will not be affected by the IF,THEN statement if the condition is false.

An ELSE statement can be included with an IF,THEN statement. An ELSE statement is used to specify a line number that the program will branch to if the control expression is false.

```
100 IF C>256 THEN 1000 ELSE 2000
```

The preceding example statement will cause the program to branch to line number 1000 if the value of the variable C is greater than 256. If the value of C is less than or equal to 256, the program will branch to line number 2000.

Branching Statements

The concept of branching a program was introduced in the discussion of conditional statements. A branch in a program is an alteration of the normal order of executing statements.

A branching statement causes the program control to proceed at a specified line number. The most commonly used branching statements are GOTO and GOSUB.

A GOTO statement has the following format.

GOTO *linenumber*

For example, the following statement would cause the program to branch to line number 1000 when line 500 is executed.

```
500 GOTO 1000
```

ON, GOTO Statement

An ON, GOTO statement is a combination of a conditional statement and a branching statement. The use of an ON, GOTO statement is illustrated in the following program.

```
10 INPUT A
20 ON A GOTO 40,50,60
40 B = B+1
50 C = C+1
60 D = D+1
```

The ON, GOTO statement at line number 20 is used to branch the program to one of the specified line numbers (40,50 or 60). The value of the variable A is used to select the line number of the branch.

If the expression following the keyword ON is equal to 1, program control will branch to the first line number specified after GOTO; if 2, to the second; if 3, to the third; etc.

For example, if the variable A is one, program control will branch to line number 40. If the value of A is 2, the program will branch to line number 50, etc.

If the control expression is not an integer, it is rounded off. If the value of the control expression is less than or equal to 0, or greater than the number of line numbers in the ON, GOTO statement, a BAD VALUE error will occur.

Subroutines & GOSUB Statements

Many times the same set of program instructions are used more than once in a program. Re-entering these program lines can be very time consuming. The use of subroutines make the additional entries unnecessary.

A subroutine can be defined as a program which appears within another larger program. The subroutine may be executed as many times as desired.

The execution of subroutines is controlled by the GOSUB and RETURN statements. The format for a GOSUB statement is as follows.

GOSUB *linenumber*

The computer will begin execution of the subroutine beginning at the specified *linenumber*. Statements will continue to be executed in order, until a RETURN statement is encountered. Upon execution of the RETURN statement, the computer will branch out of the subroutine back to the first line following the original GOSUB statement. This is illustrated in the following example.

```
                100 FOR J = 0 TO 3
                200 GOSUB 1000
                300 NEXT J
                400 END
               ⎧1000 X = J^2+J^3
Subroutine     ⎨1100 PRINT X
               ⎩1200 RETURN
```

Subroutines can make writing a program easier. By dividing a lengthy program into a number of smaller subroutines, the complexity of the program will be reduced. Individual subroutines are smaller and therefore more easily written. Subroutines are also more easily debugged than longer programs.

ON, GOSUB Statement

An ON, GOSUB statement is very similar in principle to an ON, GOTO statement. The following statement is an example of an ON, GOSUB statement.

```
100 ON X GOSUB 1000, 2000, 3000
```

If the value of X is 1, the subroutine at line 1000 is executed. If X is 2, the subroutine at line 2000 is executed. If X is 3, the subroutine at line 3000 is executed.

If the value of X is rounded off to a value less than 1 or greater than 3, a BAD VALUE error will occur.

When the execution of the subroutine is complete, program control will return to the line immediately following the ON, GOSUB statement.

Functions

Functions are used to represent a set of calculations that return a value. Generally, functions require one or more arguments. The arguments are used as input for the function. The output of the function is the value that is returned.

There are many functions that can be used in TI BASIC (See Table 2-2).

Table 2-2. TI BASIC Functions

Functions that require numeric arguments				Functions that require string arguments
ABS	COS	INT	SIN	ASC
ATN	EOF	LOG	SQR	LEN
CHR$	EXP	SGN	STR$	VAL
			TAN	

The following format is used for all of the functions that require one argument.

function (argument)

The keyword for the function is always followed by an argument. An argument can be any expression that uses constants, variables, operators, or other functions. Some functions require numeric values as arguments and others require string values. The type of data used as an argument must correspond to the type of data that the function requires. The functions on the left side of Table 2-2 require numeric arguments. The functions on the right side of the table require a string value as arguments.

A STRING-NUMBER MISMATCH error will occur if the wrong type of argument is used with a function.

The following example statements demonstrate the use of the TI BASIC functions.

```
10 PRINT SGN(SIN (A))
10 PRINT CHR$(87)
10 X = SQR (VAL(A$))
10 FOR J = LEN(A$) TO LEN(B$)
10 IF LEN(A$)>LEN(B$) THEN 80
```

Each of the TI BASIC functions are described in detail in Chapter 5.

TI BASIC includes two functions that require more than one argument. These functions are used with both numeric and string values.

The POS and SEG$ functions are the only two functions that require more than one argument.

The arguments of these functions must be enclosed in parentheses and separated by commas. The following statements are examples of multiple argument functions.

```
100 Y = POS(X$,"TEXAS",4)
100 PRINT SEG$(X$,5,6)
```

The only function that does not require an argument is RND. This function is used to return random numbers that are greater than or equal to zero and less than one.

DEF

A DEF statement can be used to define a function. If calculations in a program need to be repeated several times, a function can be defined to perform the calculations.

Functions can return either a string value or a numeric value. Also, the argument of a function can be either a string or a number.

The following example program is used to define a function that concatenates the argument of the function with the string "BERRIES".

```
>10 DEF Z$(X$) = X$ & "BERRIES"
>20 A$ = "BLUE"
>30 B$ = "STRAW"
>40 PRINT Z$(A$),Z$(B$)
> RUN
  BLUEBERRIES      STRAWBERRIES
```

The name of the function is Z$. Since the function returns a string value, the function name must be a string variable name.

The argument of the function is represented by the variable X$. In the function definition statement at line 10, the argument of the function (X$) is concatenated with the string "BERRIES".

As a result, the PRINT statement at line 40 returns two string values. The variables A$ and B$ are used as arguments of the function Z$.

The two string values returned by the function are the output of the program.

The following example program uses a DEF statement to define a numeric function. The function defined in the following program is commonly called the inverse sine or arcsine function.

```
>10 DEF ARCSIN(X) = ATN(X/SQR(-X*X+1))
>20 PRINT ARCSIN(.5)
>RUN
  .5235987756
```

ASCII

The computer cannot actually store characters in its memory. Instead, the computer stores numeric values.

Before characters can be stored, they must be converted to numbers. Computers use special numeric codes to store

characters. Most microcomputers use a code known as ASCII (American Standard Code for Information Interchange).

The TI-99/4A uses codes slightly different from the standard ASCII code set. The codes used by the TI-99/4A are listed in Appendix C.

The CHR$ function can be used to generate the characters with ASCII codes from 30 to 127. For example, since the ASCII value for the letter A is 65, the following statement causes the letter A to be output.

```
PRINT CHR$(65)
```

The ASCII values from 0 to 30 are reserved for special functions of the computer. The CHR$ function cannot be used to generate these special functions. Generally, when a CHR$ function has an argument less than 30, a blank space will be displayed.

The ASCII values from 128 to 159 are used to specify special graphics characters. These special characters must be defined before they can be displayed.

The ASC function returns the ASCII code equivalent for its string argument. If this string is longer than one character, the ASC function returns the ASCII code for the first character in the string.

The following program illustrates the use of the ASC function.

```
>100 A$ = "JOHN JOHNSON"
>200 PRINT ASC(A$)
>300 END
>RUN
```

The program in the preceding example outputs the number 74 because the ASCII value of the first character in the string argument of the ASC function is 74.

Advanced Input and Output Statements

INPUT and PRINT statements are commonly used to perform the input and output functions of the computer. However, the use of these statements is somewhat restricted. An INPUT statement can only be used to accept data from the keyboard, and a PRINT statement can only be used to output data to the display.

These statements can be modified to allow an exchange of data between the computer and the peripheral devices. The generalized I/O statements have the following format.

INPUT # *filenumber* : *data* ...
PRINT # *filenumber* : *data* ...

PRINT# and INPUT# statements are commonly used to transfer data to and from the Program Recorder or Disk Drives. PRINT# statements are also used to send output to the printer.

The use of a filenumber is the only difference between the standard and generalized I/O statements.

Filenumbers

Before communication can be undertaken between an input or output device, an I/O channel must first be associated with the specific device. The channel serves as a link between the BASIC program and the I/O device. The filenumber is a parameter that is used to specify an I/O channel.

Once an I/O channel is specified for a particular device, any input or output to the device can be performed with an INPUT# or PRINT# statement. The file number in an I/O statement must be used to specify the particular I/O channel.

OPEN

Before an I/O channel can be used with a program, it must first be opened. When a channel is opened for an external device such as the Program Recorder, disk drive or printer, the computer reserves a memory area known as a buffer from which data will be sent to or received from the specified device.

An OPEN statement has the following configuration.

OPEN # *filenumber:"device.filename"*

The *filenumber* specifies the I/O channel being opened. This number can range from 1 to 255, and will be used by subsequent INPUT# and PRINT# statements to access the device opened.

The *device* parameter is used to specify the type of I/O device being opened. The *filename* is used to specify the particular file that the input will be taken from or the output will be sent to.

PRINT# and INPUT#

PRINT# and INPUT# are used to send data to or receive data from the device opened in a preceding OPEN statement.

The exact nature of the use of these generalized I/O statements is described in the chapters of this book that deal with the specific I/O devices.

CLOSE

A CLOSE statement is used to prevent access to a device that was previously accessed with an OPEN statement.

EDIT

The edit mode is a feature of the TI-99/4A that allows the statements in a program to be revised. The edit mode allows program lines to be modified without retyping the entire statement.

The EDIT command can be used to enter the edit mode. The EDIT command must be followed by the line number of the statement that needs to be modified. The following commands can be used to modify a program line when the edit mode is in effect.

FCTN ↑	FCTN 1 (DEL)
FCTN ↓	FCTN 2 (INS)
FCTN ←	FCTN 3 (ERASE)
FCTN →	

CHAPTER 3. PROGRAMMING IN TI EXTENDED BASIC

TI Extended BASIC is a programming language that can be used with the TI-99/4A. Unlike the standard version of TI BASIC, Extended BASIC is not built into the computer. Extended BASIC can be used only if the TI Extended BASIC solid state cartridge is plugged into the command module slot in the computer console.

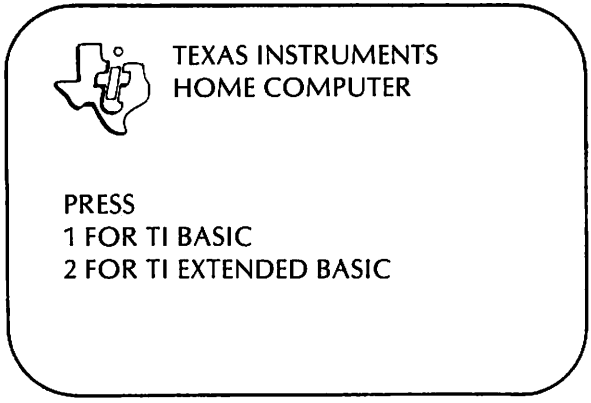
The TI Extended BASIC cartridge is not included with the TI-99/4A computer. In order to use Extended BASIC, the cartridge must be purchased separately.

This chapter is a presentation of the programming features of Extended BASIC. Since Extended BASIC is an expanded version of TI BASIC, the information in Chapter 2 applies to Extended BASIC as well as TI BASIC.

Since this chapter is based on the information in chapter 2, please read chapter 2 before reading this chapter unless you are already familiar with TI BASIC.

Only one of the two versions of BASIC can be used at any time. As a result, the version must be selected when the computer is powered on.

When the computer is powered on, the master title screen will be displayed. When any key on the keyboard is pressed, the main selection menu will be displayed. When the Extended BASIC cartridge is being used with the computer, the main selection menu will appear as follows.



In order to use Extended BASIC, press the 2 key on the keyboard. When the 2 key is pressed, the display will be cleared and the following message will appear on the display.

```
* READY *
```

The "greater than" symbol (>) is used to indicate that the computer is ready to accept a command.

Multiple Statement Lines

One of the principle differences between TI BASIC and Extended BASIC is the use of multiple statement lines. In Extended BASIC, more than one statement can be included on each program line. The individual statements on each program line must be separated by two colons. For example, the following program line contains 3 statements.

```
10 FOR J = 1 TO 10::PRINT A(J)::NEXT J
```

This feature allows programs to be written with fewer program lines.

REM statements should be the last statement in a multiple statement program line. When a REM statement is encountered in a program, any subsequent statements on the same program line will be ignored.

Each DATA statement in a program must be the only statement on a program line.

Variable Assignment Statements

In Extended BASIC two additional statements are available for assigning values to variables. These statements are ACCEPT and LINPUT. Also, the LET statement has additional features when used in Extended BASIC.

ACCEPT

An ACCEPT statement is used to assign values to variables while a program is being executed.

An ACCEPT statement can only be used to input data that is entered via the keyboard.

An ACCEPT statement is similar in principle to an INPUT statement. However, an ACCEPT statement has more optional features than does an INPUT statement.

ACCEPT statements can include any or all of the following options.

AT VALIDATE BEEP ERASE ALL SIZE

The AT option allows the data being input to be displayed at any specified location on the display.

The VALIDATE option can be used to restrict the type of data that can be input.

The VALIDATE option can be used to restrict the input to any combination of the following catagories:

1. Upper case letters (UALPHA)
2. Digits 0 through 9 (DIGIT)
3. Numeric values (NUMERIC)
4. Specific characters (e.g. "YN")

The BEEP option causes a tone to sound when the ACCEPT statement is executed.

The ERASE ALL option causes the entire display to be cleared before any data is input.

The SIZE option is used to restrict the number of characters that can be included in a value being input.

The following example statements demonstrate the use of ACCEPT statements:

```
10 ACCEPT AT (1,1) BEEP SIZE(10):X
20 ACCEPT VALIDATE (DIGIT) ERASE ALL:Y
```

The first example statement causes the data being input to appear at the upper left corner of the display. A tone is sounded when the statement is executed. The maximum number of characters that can be included in the input is 10. The value that is input at the keyboard will be assigned to the variable X.

The second example statement causes the data being input to be assigned to the variable Y. The VALIDATE option allows the data being input to consist only of the digits 0 through 9. When the statement is executed, the screen will be cleared before any data is accepted.

If any characters other than the digits 0 through 9 are entered as data, a tone will sound, and the input will not be accepted.

LINPUT

A LINPUT statement is used in Extended BASIC to assign a value to a string variable. A LINPUT statement is similar to an INPUT statement, but a LINPUT statement can only assign a value to one variable.

Since a LINPUT statement can only assign a value to one variable at a time, a comma is not used to separate data items. As a result, commas are considered a part of the string value. In other words, a LINPUT statement is used to assign an entire line of data to a string variable.

The following example demonstrates the use of a LINPUT statement.

```
> 10  LINPUT A$
> 20  PRINT A$
> RUN
  ?  PROGRAM LINES 20, 22, 28-32 ←— user's response
  PROGRAM LINES 20, 22, 28-32
```

The value that is entered in response to the LINPUT statement contains two commas. These commas are not used to separate data items. Instead, the commas are considered part of the string data.

A LINPUT statement can include a prompt message that appears each time the LINPUT statement is executed.

The following example consists of a LINPUT statement that contains a prompt message.

```
10 LINPUT "ENTER NAME (LAST, FIRST)":NAME$
```

The preceding example statement would cause the following message to be displayed each time the statement is executed.

```
ENTER NAME (LAST, FIRST)
```

A LINPUT statement can also be used to input data from a device other than the keyboard. If a LINPUT statement includes a file number, the data will be taken from the device that was previously opened for input.

If a LINPUT statement includes a filenumber, the filenumber must correspond to the file number specified in a previously executed OPEN statement. The configuration of a LINPUT# statement is as follows.

LINPUT # *filenumber* : *variable*

LET

In Extended BASIC, a variable assignment statement can be used to assign a value to several variables. Each variable in a multiple variable assignment statement must be separated by a comma. The following example statement would assign the value 100 to the variables X, Y and Z.

```
100 X, Y, Z = 100
```

Arrays

Subscripted variables can be used to make the handling of variables easier. Extended BASIC allows variables to be used with up to seven subscripts.

Due to the large amount of memory required for a multi-dimensional array, large arrays must be used carefully.

The following example program uses a four dimensional array.

```
>10 DIM A (2, 3, 4, 5)
>20 A (1, 1, 1, 1) = 10
>30 PRINT A (1, 1, 1, 1)
>RUN
  10
```

A four dimensional array can be visualized as a number of books that contain data. If each book has data arranged in rows and columns, any data item in any of the books can be specified with 4 subscripts.

The first subscript refers to the book number. The second subscript refers to page number. The third and fourth subscripts refer to the row and column number of the data item.

Boolean Operators

In addition to the arithmetic, relational and string operators, Extended BASIC allows the use of Boolean (or logical) operators.

Boolean operators are used to compare logical expressions. Logical expressions are any expressions that can be considered either true or false.

The most common type of logical expressions are relational expressions. These expressions are always either true or false.

The four Boolean operators used with the TI-99/4A are AND, OR, XOR and NOT. The AND, OR and XOR operators require two arguments.

The AND operator returns a true value only if both of the arguments are true. The OR operation returns a true value if either one (or both) of the arguments are true. The XOR operator returns a true value if either of the arguments (but not both) are true.

The NOT operator requires only one argument. This operator returns a true value only when the argument is false. Illustration 3-1 demonstrates the results of the Boolean operators.

The Boolean operators are most commonly used in IF, THEN statements. The following statements demonstrate the use of the Boolean operators.

```
100 IF X >10 OR Y < 10 THEN 250
200 IF A$ = "STOP" AND Z = 10 THEN 500
100 IF NOT Y = 8 AND X = 200 THEN 50
```

Illustration 3-1. Boolean Operators.

X	Y	X OR Y
T	T	T
T	F	T
F	T	T
F	F	F

X	Y	X AND Y
T	T	T
T	F	F
F	T	F
F	F	F

X	Y	X XOR Y
T	T	F
T	F	T
F	T	T
F	F	F

X	NOT X	
T	F	
F	T	

Formatted Output

Extended BASIC allows data to be output according to a predefined format. There are three statements that can be used to generate formatted output: PRINT USING, DISPLAY USING and IMAGE.

Formatted output requires a predefined pattern that is used to output the data. The pattern that is used for the data is defined with a **format string.** A format string is a set of characters that describe the format of output data.

A format string can be used to specify the maximum number of characters that can appear in an output value. A format string can also be used to indicate the position of the decimal point within the value. Furthermore, a format string can be used to specify that a value should be output in scientific notation.

There are three characters that can be used in a format string. These characters are the pound sign (#), caret (^) and decimal point (.). The pound sign is used to represent a character in the value being output. The decimal point is used to specify the location of the decimal point within the value. The carets in a format string are used to indicate that a value should be output in scientific notation.

Some examples of format strings are presented in Illustration 3-2.

Illustration 3-2. Format Strings

DATA	FORMAT STRING	OUTPUT
23.478	##.##	23.48
4	#.###	4.000
24.8785	##.###^^^^	2.488E + 01

A format string can be used to specify the format of string values as well as numeric values.

A format string can be used in one of two ways. An IMAGE statement can include a format string, and each PRINT using or DISPLAY using statement that requires the specified format can use the line number of the IMAGE statement. This concept is demonstrated in the following example.

```
> 100 INPUT X
> 200 IMAGE ##.##
> 300 DISPLAY USING 200: X
> RUN
  ? 24.685 ←— user's response
   24.69
```

An alternate way of using a format string is to include the format string in the PRINT or DISPLAY statement that requires the specified format.

```
> 100 INPUT X
> 300 DISPLAY USING "##.##": X
> RUN
    ? 24.685 ←— user's response
   24.69
```

A format string can include characters other than the characters (#)(^) and (.). However, any of these characters appear in the output exactly as they appear in the format string.

The following example includes a format string that contains characters other than the special formatting characters.

```
200 PRINT USING "PART NUMBER #####": X
```

Typical output of the preceding example statement would appear as follows

```
PART NUMBER 98563
```

Notice that numeric values are rounded off in order to be displayed with the specified number of decimal places. If a value is too large to be displayed in the specified format, asterisks will be displayed instead of the specified value.

```
> 10 DISPLAY USING "##.#": 250
> RUN
  ****
```

Since the value 250 cannot be displayed with two digits to the left of the decimal point, the preceeding example program outputs four asterisks instead of the value 250.

The following example demonstrates the use of formatted string output.

```
> 10 INPUT A$
> 20 IMAGE PLEASE CONTACT ###########
> 30 PRINT USING 20: A$
> RUN
  ? JOE SMITH ← user's response
  PLEASE CONTACT JOE SMITH
```

String values that are output according to a format string are always left justified. However, numeric values are always right justified. Numeric values that include a decimal point are aligned according to the position of the decimal point.

Illustration 3-3 demonstrates left justified, right justified and aligned output.

Illustration 3-3. Justification of Output

FORMAT	#####	#####	##.##
JUSTIFICATION	**left**	**right**	**aligned**
OUTPUT	BILL	7	7.00
	BOB	1238	33.85
	RANDY	48	8.40
	TOM	541	77.85

BRANCHING STATEMENTS

Extended BASIC allows the use of several branching statements that TI BASIC does not allow. Extended BASIC also allows more features of the IF, THEN, ELSE statement.

In TI BASIC, an IF, THEN, ELSE statement can only be used with specific line numbers. In extended BASIC, however, statements can be used instead of line numbers.

For example, the following IF, THEN statement contains two statements instead of two line numbers

```
400 IF X>256 THEN T=0 ELSE T=-1
```

The preceding example uses the expression X>256 to control the execution of the program. If the expression is true, the T=0 statement will be executed. If the expression is false, the T=-1 statement will be executed.

An IF, THEN, ELSE statement can also include multiple statements. In order for the statements to be executed as a single statement, the individual statements must have a double colon (::) between them.

For example, the following statement will assign values to the variables J and K if the control expression is true. Values will be assigned to the variables L and M if the control expression is false.

```
10 IF X>10 THEN J=10::K=11 ELSE L=-4::M=-5
```

ON BREAK

An ON BREAK statement is used to determine the action that will be taken when a breakpoint is encountered in a program. A breakpoint is an interruption in the execution of a program due to a BREAK statement or the CLEAR command (FCTN 4).

When the keyword NEXT is used in an ON BREAK statement, all the breakpoints in a program will be ignored.

An ON BREAK statement that uses the keyword STOP corresponds to the default mode of the computer. When this mode is in effect, the program execution will stop when a breakpoint is encountered.

An ON BREAK statement can only have one of the following two formats.

```
ON BREAK NEXT
ON BREAK STOP
```

ON ERROR

An ON ERROR statement is used to determine the action that will be taken when an error occurs during the execution of a program.

When a line number is included in an ON ERROR statement, program control will branch to the specified line number when an error occurs. The following example statement is a typical ON ERROR statement.

```
200 ON ERROR 1000
```

A RETURN statement can be used to return the program control to the statement that caused the error.

A RETURN NEXT statement can be used to branch program control to the statement that follows the statement that caused the error.

If a RETURN statement includes a line number, program control will be branched to the specified line number.

The following four techniques can be used to branch a program when an error occurs.

1. Branch the program to a specified line number when an error occurs.

2. Branch the program to a specified line number when an error occurs. Proceed by executing one or more program statements and use a RETURN statement to branch program control back to the statement that caused the error.

3. Branch the program to a specified line number when an error occurs. Proceed by executing one or more program statements and use a RETURN statement (with a line number) to branch program control to the specified line number.

4. Branch the program to a specified line number when an error occurs. Proceed by executing one or more program statements. Then, use a RETURN NEXT statement to branch program control to the statement that follows the statement that caused the error.

An ON ERROR statement can also include the keyword STOP. This statement causes the execution of a program to stop when an error occurs. Since this statement corresponds to the default mode of the computer, an ON ERROR STOP statement is generally used only to counteract a preceeding ON ERROR statement.

ON WARNING

An ON WARNING statement determines the action that will be taken when a warning condition develops. An ON WARNING statement can include any one of the following three keywords.

STOP PRINT NEXT

An ON WARNING STOP statement causes the execution of the program to stop when a warning condition occurs.

An ON WARNING PRINT statement causes an appropriate warning message to be displayed when a warning condition occurs. This statement corresponds to the default mode of the computer.

An ON WARNING NEXT statement causes the warning conditions to be ignored by the computer. Warning messages are not diplayed when an ON WARNING NEXT statement is in effect.

SUBPROGRAMS

Extended BASIC allows the use of **subprograms.** Subprograms are similar to subroutines, but subprograms are accessed with CALL statements. As a result, it is not necessary to branch program control to a subprogram.

Subprograms are similar to functions, but subprograms are not restricted to accepting only one input value. Furthermore, subprograms can be used to compute values for any number of variables. Subprograms can also be used to call other subprograms.

There are many pre-defined subprograms that can be used in Extended BASIC. Each of these subprograms are described in detail in chapter 5. The CALL statement is used to access each of these subprograms.

The subprograms that have been pre-defined are commonly used for generating sound and graphics.

SUB

The SUB statement allows subprograms to be written with BASIC statements. A SUB statement must include the subprogram name as well as a list of variables that are used in the subprogram.

The list of variables that are included in a SUB statement are used throughout the subroutine. However, these variables are considered separate from the variables in the main program. The

variables that are assigned values in the subprogram do not necessarily represent the same values in the main program. For example, the variable X may be assigned the value 10 in the main program. Later, the varibale X may be assigned the value 20 in a subprogram. As a result, the variable X will be assigned two different values at the same time.

Since this situation is potentially very confusing, do not duplicate the main program variables in a subprogram.

The following example demonstrates the use of a subprogram.

```
Main      >10 A=27
Program   >20 B=34
          >30 C=54
          >40 CALL ADD (A,B,C)
          >50 END
Sub-      >100 SUB ADD (X,Y,Z)
Program   >110 SUM=X+Y+ Z
          >120 PRINT SUM
          >130 SUBEND
          >RUN
            115
```

Subroutines must be defined at the end of the main program. If the program contains more than one subprogram, the subprograms must be defined one after the other at the end of the main program. Be sure to include all of the main program statements (including DATA statements) before the subprogram definitions.

The example program contains a subprogram called ADD. The statements from line number 100 to 130 are used to define the subprogram. Note that the subprogram definition begins with a SUB statement and ends with a SUBEND statement.

The variables A, B, and C are used in the main program. The variables X, Y, Z and SUM are used only within the subprogram.

Values are assigned to the variables A, B and C in the main program. The CALL statement at line number 40 is used to access the subprogram ADD. The values of the variables A, B and C are passed to the subprogram variables X, Y and Z. The calculations in the subprogram are performed with the values of the variables A, B and C used in place of the variables X, Y and Z.

When the example program is executed , the value 115 is output. This valve is the sum of the values of the variables A, B and C.

FUNCTIONS

Extended BASIC allows the use of 5 functions that cannot be used in TI BASIC. The additional functions are described briefly in Table 3-1.

Table 3-1. Extended BASIC Functions

MAX (a,b)	Returns the greater of the two arguments.
MIN (a,b)	Returns the lesser of the two arguments.
PI	Returns the value 3.141592654.
REC (a)	Returns the current record number of a relative file.
RPT$ (A$,a)	Returns a specified number of repetitions of the specified string value.

MANIPULATING PROGRAMS

The RUN command can be used in Extended BASIC to load and execute a program. With a disk drive, the RUN command can be used to load and execute any program that is stored on a

diskette. With a program recorder, the RUN command can be used to load and execute a program that was previously stored on a cassette tape.

The RUN command can have either of the following two configurations.

RUN *"device"*
RUN *"device. Program name"*

Program names are only used with programs that are stored on a diskette.

Extended BASIC also allows a RUN statement to be used in a program. This feature allows the program that is currently being executed to load and execute another program.

Consider the following two programs that have been saved in diskette program files.

```
PROGRAM1

10 FOR J=1 TO 100
20 PRINT J
30 NEXT J
40 RUN "DSK1.PROGRAM2"

PROGRAM2

10 FOR J=100 TO 1 STEP -1
20 PRINT J
30 NEXT J
40 END
```

Line number 40 in PROGRAM1 is a RUN statement that causes PROGRAM2 to be loaded and executed. As a result, the following statement can be used to load and execute PROGRAM1, which in turn loads and executes PROGRAM2.

RUN "DSK1. PROGRAM1"

The first program causes the numbers from 1 to 100 to be output. When the first program is completed, the computer will pause briefly while the second program is loaded. The second program causes the numbers from 1 to 100 to be output in descending order.

A RUN statement can be used to execute programs that are stored on cassette tape. However, a program recorder is not as convenient to use as a disk drive.

CHAPTER 4
TI SOUND & GRAPHICS

This chapter provides an overview of the various sound and graphics capabilities of the TI-99/4A computer when used with Standard BASIC and Extended BASIC.

TI SOUND

The TI-99/4A has the capability to generate a wide variety of sounds. Sounds are transmitted directly through the speaker in the television set or monitor.

All sound capabilities can be controlled with a single statement. The CALL SOUND statement is used to activate the sound generators built into the computer.

Generating Sound

The CALL SOUND statement is used to output sound via the television set or monitor. CALL SOUND is used with the following configuration.

CALL SOUND *(delay, frequency 1, volume 1, [. . . frequency 4, volume 4])*

The arguments of the CALL SOUND statement determine the type, volume, and duration of the sound or sounds to be generated. A maximum of three tones and one noise can be generated simultaneously in one CALL SOUND statement.

The first argument of the CALL SOUND statement is the *delay*. The *delay* determines the duration of the specified sound or sounds. Only one *delay* may be specified in a CALL SOUND statement.

The delay is measured in milliseconds and can range anywhere from 1 to 4250 milliseconds. A sound will continue to be generated until the specified duration has been completed.

If a CALL SOUND statement is encountered while a previous CALL SOUND statement is being executed, program execution will pause until the first CALL SOUND statement has been completed. For example, the following program will complete execution of the first CALL SOUND statement before continuing with the second.

```
100 CALL SOUND (3000, 110, 0)
200 CALL SOUND (3000, 400, 0)
```

If a negative sign is inserted in front of the first parameter *(delay)* the specified tone will be generated immediately. For example, the following program is similar to the previous program except for the minus sign that was inserted before the first parameter in the second CALL SOUND statement. When this program is executed, the first CALL SOUND statement will be interrupted by the second CALL SOUND statement.

```
100 CALL SOUND (3000, 110, 0)
200 CALL SOUND (-3000, 400, 0)
```

The second parameter of the CALL SOUND statement is the *frequency*. The *frequency* specifies the tone or noise to be generated. The *frequency* can range anywhere from 110 to 44733 or from -8 to -1. The values 110 through 44733 specify musical notes. The value 110 specifies the lowest possible note that the computer can generate while 44773 specifies the highest.

The computer can generate eight different noises. Each noise is represented by a number from -8 to -1. The best way to become familiar with these noises is to listen to each one separately.

The third parameter of the CALL SOUND statement specifies the volume at which each note is to be generated. The specified *volume* must be an integer from 0 to 30. A *volume* of 0 causes the specified sound to be generated as loud as possible. A *volume* of 30 causes the specified sound to be generated as quietly as possible.

Generating Chords

The CALL SOUND statement can be used to generate two or three notes simultaneously. The following configuration is used to generate more than one note.

CALL SOUND (*delay, frequency 1, volume 1, frequency 2, volume 2, frequency 3, volume 3*)

Only one *delay* parameter may be used in a CALL SOUND statement. Therefore, all the specified notes will be generated for the same length of time. Up to three notes can be generated simultaneously, but each note requires both a *frequency* and a *volume*. The following example CALL SOUND statement demonstrates this principle.

```
CALL SOUND (4000, 196, 0, 262, 0, 330, 0)
```

One noise can also be generated along with one, two, or three notes. The following example is similar to the previous one, except that a noise was added.

```
CALL SOUND (4000, 196, 0, 262, 0, 330, 0, -7, 0)
```

A total of four different sounds can be generated with one CALL SOUND statement. However, the following restrictions apply:

1) Not more than three notes may appear in one CALL SOUND statement.
2) Not more than one noise may appear in one CALL SOUND statement.
3) Each specified sound must be followed by a volume.
4) All sounds specified in a single CALL SOUND statement are generated for the same amount of time as specified by the first parameter.

Using The CALL SOUND Statement

The CALL SOUND statement can be used in either the command mode or in the program mode.

When the CALL SOUND statement is used in the command mode, the cursor will return almost immediately after pressing ENTER. The specified sound will continue to be generated until the specified duration has been completed.

Operation of the computer does not pause while sound is being generated. As a result, commands may be entered while a sound is being generated.

The CALL SOUND statement works in a similar manner in the program mode. Execution of a program will not stop while sound is being generated. The following program shows that even while a sound is being generated, the computer will continue executing the program.

```
100 CALL SOUND (4250, 500, 0)
200 PRINT "EXECUTION IS COMPLETE"
```

If more than one CALL SOUND statement is used within a program, a subsequent CALL SOUND statement will not be executed until the current CALL SOUND statement has been completed. However, if a negative sign is inserted before the *delay* parameter in a CALL SOUND statement, the specified sound will be generated immediately.

The following program shows the effects of using a *delay* parameter with and without a negative sign.

```
100 CALL SOUND (4000, 300, 0)
200 PRINT "PHASE #1"
300 CALL SOUND (4000, 600, 0)
400 PRINT "PHASE #2"
500 CALL SOUND (-4000, 900, 0)
600 PRINT "PHASE #3"
```

Using Variables In CALL SOUND Statements

The parameters of the CALL SOUND statement can include numeric variables, numeric expressions, and numeric functions.

The following program includes a CALL SOUND statement that uses numeric variables and expressions as its parameters.

```
100 FOR T=1 TO 10
200 CALL SOUND (1000, 110 + (10*T), T)
300 NEXT T
```

Functions that return numeric values can also be used as parameters within a CALL SOUND statement. The following program will generate ten random notes.

```
100 FOR T=1 TO 10
200 CALL SOUND (1000 RND*891 + 110, 0)
300 NEXT T
```

TI GRAPHICS

In this section, as well as the remainder of this chapter, we will explain the various graphic capabilities of the TI-99/4A computer when used with Standard and Extended BASIC.

Standard Character Set

The TI-99/4A has a predefined character set. This character set consists of uppercase and lowercase letters. Also, any special characters such as +, –, =, ., ;, etc., are also predefined. All predefined characters have a corresponding number known as the ASCII code of that character. A table of the predefined characters and their ASCII codes are listed in Appendix C.

Redefining A Character

Characters that are predefined can be temporarily redefined as another character. A CALL CHAR statement can be used to redefine any or all of the characters in the character set.

A CALL CHAR statement has the following configuration.

CALL CHAR *(ASCII code, "string expression")*

The first parameter is an integer from 32 through 159* that indicates the ASCII code of the character to be redefined. The second parameter specifies the new character. The *string expression* is the hexadecimal representation of the new character.

The Hexadecimal Conversion

Hexadecimal notation is a compressed method of describing binary numbers. Combinations of four binary digits can be described using a single hexadecimal character.

Table 4-1 illustrates the conversion from binary to hexadecimal notation. Every combination of four binary digits can be expressed with a single hexadecimal character.

Table 4-1. Binary to Hexadecimal Conversion

BINARY	HEX
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

BINARY	HEX
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

The Size of One Character

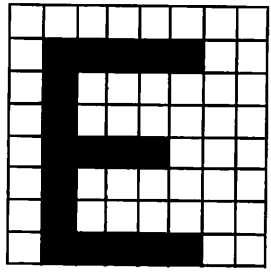
One character is made up of 64 picture elements (or pixels), which are arranged in an 8x8 matrix. The character is created by illuminating only certain pixels, thus causing a shape to appear.

---

*Although the CALL CHAR statement can be used in both Standard BASIC and Extended BASIC, only characters 32 through 143 can be redefined in Extended BASIC.

Illustration 4-1 shows an enlarged view of the 64 pixels that make up one character. This illustration also shows how a character can be created by illuminating only certain pixels.

Illustration 4-1. Illuminating Pixels

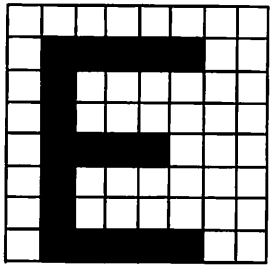


USING HEXADECIMAL NOTATION TO DESCRIBE A CHARACTER

Each character consists of an 8 x 8 matrix of pixels that can be described in terms of binary digits. Every illuminated pixel can be described by the binary digit 1. Every non-illuminated pixel can be described by the binary digit 0. Therefore, the description of each character requires 64 binary digits.

Illustration 4-2 depicts a typical character and its binary representation.

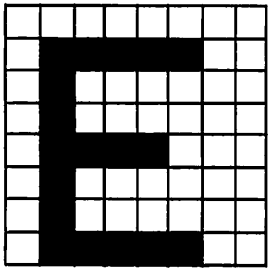
Illustration 4-2. Binary Representation of a Character



```
0 0 0 0 0 0 0 0
0 1 1 1 1 1 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 1 1 1 0 0 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 1 1 1 1 1 0 0
```

Since each group of 4 binary digits is equivalent to a single hexadecimal character, the 64 digit binary code can be replaced by a 16 digit hexadecimal code. Illustration 4-3 contains a complete example of the procedure used to determine the hexadecimal representation of a character.

Illustration 4-3. Character Definition and Conversion



binary representation	hexadecimal representation
0 0 0 0 0 0 0 0	0 0
0 1 1 1 1 1 0 0	7 C
0 1 0 0 0 0 0 0	4 0
0 1 0 0 0 0 0 0	4 0
0 1 1 1 1 0 0 0	7 8
0 1 0 0 0 0 0 0	4 0
0 1 0 0 0 0 0 0	4 0
0 1 1 1 1 1 0 0	7 C

Using the CALL CHAR Statement

As stated earlier, the CALL CHAR statement can be used to create new characters. The configuration of the CALL CHAR statement is as follows:

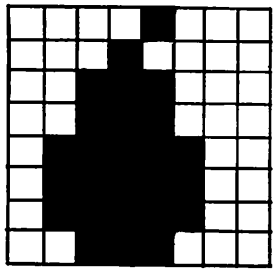
CALL CHAR *(ASCII code, "hexadecimal representation")*

The first parameter is the ASCII code of the character to be redefined. The second parameter is the hexadecimal representation of the new character. When the CALL CHAR statement is executed, the new character will be assigned to the specified ASCII code.

Illustration 4-4 shows how the CALL CHAR statement can be used to create a new character. First, the new character must be drawn in an 8 x 8 matrix. Next, the hexadecimal representation of the character must be determined. Finally, the hexadecimal code, along with the ASCII code of the character to be replaced

must be used in a CALL CHAR statement. Whenever the specified ASCII code is used in a PRINT, DISPLAY, CALL VCHAR or CALL HCHAR statement, the new character will appear.

Illustration 4-4. Creating A Character



binary	hexadecimal
0 0 0 0 1 0 0 0	0 8
0 0 0 1 0 0 0 0	1 0
0 0 1 1 1 0 0 0	3 8
0 0 1 1 1 0 0 0	3 8
0 1 1 1 1 1 0 0	7 C
0 1 1 1 1 1 0 0	7 C
0 1 1 1 1 1 0 0	7 C
0 0 1 1 1 0 0 0	3 8

```
100 CALL CHAR (130, "081038387C7C7C38")
200 FOR T = 1 TO 28
300 PRINT CHR$(130);
400 NEXT T
```

When the program in Illustration 4-4 is executed, a row of pears will be displayed on the screen.

Program and Immediate Modes

The characters with ASCII codes from 132 to 159* can be redefined at any time. These characters will retain their new definition until the computer is turned off or BASIC is exited. The characters with ASCII codes from 32 to 131** can also be redefined at any time. However, the new definition will only be retained as long as a program is being executed.

---

* 127 to 143 in Extended BASIC
** 30 to 126 in Extended BASIC

Using Variables In The CALL CHAR Statement

Both parameters of the CALL CHAR statement may be variables that had been predefined. The variable used for the ASCII code must be a numeric variable. The variable used for the hexadecimal code, however, must be a string variable.

The following statement is a typical example of the syntax of a CALL CHAR statement.

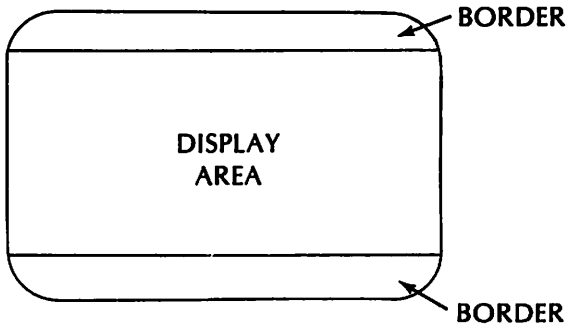
```
CALL CHAR (A, C$)
```

Functions that return the proper values may also be used within a CALL CHAR statement. The ASC function, for example, may be used in place of the first parameter.

Placing A Character On The Screen

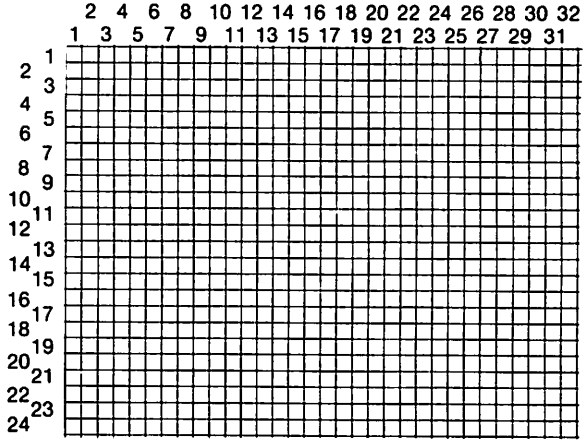
The screen is divided into two areas, the display area and the border. Characters cannot be placed in the border area.

Illustration 4-5. Screen Display



The display area is divided into 24 rows with 32 columns each. Therefore, the display area is divided into 768 separate locations. One character can be placed in each of these. Illustration 4-6 shows the division of the display area.

Illustration 4-6. Display Area



Several statements can be used to place a character on the display area. The PRINT and DISPLAY statements can be used along with the CHR$ function to place a character at the bottom, left corner of the screen (column 3, row 23). The following PRINT statement uses the CHR$ function to display a character.

```
PRINT CHR$(65)
```

The PRINT and DISPLAY statements are limited in that they can place a character in only one position (column 3, row 23). However, CALL HCHAR and CALL VCHAR statements can be used to place a character in any specified position.

The CALL HCHAR and CALL VCHAR statements can be used to place a character at a specified position on the screen. Both statements can also be used to repeat the specified character a specified number of times. The CALL HCHAR statement can be used to repeat a character horizontally and the CALL VCHAR statement can be used to repeat a character vertically.

All of the aforementioned methods for displaying a character use the ASCII code of the character. For example, the statement PRINT CHR$(65) would display the character that corresponds to the ASCII code 65. The same format applies to the DISPLAY statement.

The CALL HCHAR and CALL VCHAR statements are more versatile than PRINT and DISPLAY. Both the CALL HCHAR and CALL VCHAR statements have up to four parameters. These four parameters specify the character to be displayed, the position in which the character is to be placed, and the number of times the specified character is to be repeated.

The configurations of the CALL HCHAR and CALL VCHAR statements are as follows:

CALL HCHAR *(row, column, ASCII code, [repeat])*
CALL VCHAR *(row, column, ASCII code, [repeat])*

The parameters of the two statements are identical. The first two parameters specify the position at which the character is to be placed (see Illustration 4-6). The third parameter specifies the ASCII code of the character to be displayed. The fourth parameter specifies the number of times the character is to be repeated.

The CALL HCHAR and CALL VCHAR statements differ only when the optional fourth parameter is used. The CALL HCHAR statement will repeat the character horizontally. The CALL VCHAR statement will repeat the character vertically.

The CALL HCHAR and CALL VCHAR statements can use numeric variables, functions, and expressions as their arguments. The following program shows a CALL HCHAR statement that uses variables, functions, and expressions. The program results in a pyramid of x's being formed.

```
100 CALL CLEAR
200 FOR A = 1 TO 16
300 CALL HCHAR (24-A, A, ASC ("x"), 32-2*A)
400 NEXT A
```

Moving a Character

There is no single statement in Standard BASIC that allows a character to move along the screen. However, by continually erasing a character and placing it in a new position, a character will appear to move. The following program illustrates this method.

```
100 CALL CLEAR
200 FOR T = 1 TO 32
300 CALL HCHAR (10, T, ASC ("O"))
400 CALL CLEAR
500 NEXT T
600 GOTO 200
```

The preceding program will place a character in a screen position. The screen will then be cleared and the character will be placed in the next column of the same row. This procedure will continue until the character reaches column 32. At this point, program control branches back to the beginning.

Coloring a Character

Characters cannot be colored individually. Instead, groups of several characters are colored at the same time. To color a character, the entire group that contains that specific character must be colored as well.

Each group is assigned a group number. There are 16 different groups in Standard BASIC and 15 different groups of Extended BASIC. Table 4-2 lists the groups and their corresponding group numbers in Standard BASIC. Table 4-3 lists the groups and their corresponding group numbers in Extended BASIC. When specifying a group of characters, the group number must be used.

Table 4-2. Color Groups Standard BASIC

group number	ASCII codes
1	32-39
2	40-47
3	48-55
4	56-63
5	64-71
6	72-79
7	80-87
8	88-95

group number	ASCII codes
9	96-103
10	104-111
11	112-119
12	120-127
13	128-135
14	136-143
15	144-151
16	152-159

Table 4-3. Color Groups Extended BASIC

group number	ASCII codes
0	30-31
1	32-39
2	40-47
3	48-55
4	56-63
5	64-71
6	72-79
7	80-87

group number	ASCII codes
8	88-95
9	96-103
10	104-111
11	112-119
12	120-127
13	128-135
14	136-143

The CALL COLOR Statement

The CALL COLOR statement is used to color a group of characters. The configuration of the CALL COLOR statement is as follows:

CALL COLOR *(group number, character color, screen color)*

The first parameter of the CALL COLOR statement specifies the group of characters to be colored (see Tables 4-2 and 4-3). The second parameter specifies the color of the character. The third parameter specifies the color of the screen directly behind the character.

The second and third parameters of the CALL COLOR statement must be integers between 1 and 16 inclusive. Each one of these numbers represents a particular color. Table 4-4 lists the 16 possible colors along with their corresponding number.

The following statement is a typical CALL COLOR statement.

```
CALL COLOR (6, 5, 4)
```

This CALL COLOR statement will color all the characters in group 6. The colors of the characters will be dark blue (code 5) and the color of the screen behind those characters will be light green (code 4).

Table 4-4. Color Codes

COLOR CODE	COLOR	COLOR CODE	COLOR
1	Transparent	9	Medium Red
2	Black	10	Light Red
3	Medium Green	11	Dark Yellow
4	Light Green	12	Light Yellow
5	Dark Blue	13	Dark Green
6	Light Blue	14	Magenta
7	Dark Red	15	Gray
8	Cyan	16	White

The parameters of the CALL COLOR statement may be numeric variables, numeric functions, or numeric expressions. The only restriction is that the values must be in the allowed range.

Screen Color

The color of the entire screen can be changed by one statement. The CALL SCREEN statement can be used to change the color of the screen to any specified color.

The CALL SCREEN statement has only one parameter. This parameter is the number (1-16) that corresponds to the desired color. A table of colors and their corresponding codes can be found on the preceding page.

The CALL SCREEN statement can be used in both the command mode and program mode. However, when used in the command mode, the screen will only change colors briefly. When used in the program mode, the screen will remain the specified color until a new CALL SCREEN statement is executed, or until the program ends.

The parameter of the CALL SCREEN statement can be a numeric variable, function or numeric expression.

A typical CALL SCREEN statement is shown below.

```
CALL SCREEN (7)
```

The previous statement will change the color of the screen to dark red.

Clearing The Screen

The entire screen can be cleared with the CALL CLEAR statement. The CALL CLEAR statement can be used either in a program or in the command mode.

Locating A Character

The CALL GCHAR statement can be used to determine the character that is located at a specified position. The CALL GCHAR statement requires three parameters. The first two parameters are the coordinates of the screen position. The third parameter is a numeric variable. The ASCII code of the character located at the specified position will be assigned to the specified variable.

```
CALL GCHAR (4, 7, X)
```

The previous statement will assign the ASCII code of the character located in row 4 of column 7 to the variable X.

EXTENDED BASIC GRAPHICS FEATURES

All the capabilities that have been discussed thus far are common to both Standard BASIC and Extended BASIC. The remainder of this chapter will deal with capabilities found only in Extended BASIC.

SPRITES

The major advantage of Extended BASIC over Standard BASIC is the ability to create **sprites.** Sprites are graphic characters that can be made to move smoothly across the screen. Sprites can be enlarged up to 16 times the normal size of a character, Also, the position, the appearance, the speed, and many other features of a sprite can be altered with individual statements.

Rules of Sprites

The following are a set of rules that pertain to sprites. The following rules hold when using sprites in any situation.

1) A total of 28 sprites may appear on the screen at any one time. As a result, each sprite must be identified by a number from 1 to 28.

2) No more than four sprites may appear in one row. If more than four sprites are positioned in a single row, only the four sprites with the lowest sprite numbers will be visible.

3) Sprites will cease to exist when the execution of the program is complete or when an error occurs.

4) Once a sprite is set into motion, it will continue in that direction until a statement changes its movement or the program ends.

6) If two or more sprites are located in the same position, the sprite with the lowest sprite number will cover the other sprites. Sprites are always visible when they pass over fixed characters on the display.

Creating A Sprite

A sprite can be created in a single statement. The CALL SPRITE statement is used to create a sprite. The configuration of the CALL SPRITE statement is as follows:

CALL SPRITE (#*sprite number, ASCII-code, color, row-pixel, column-pixel* [, *row-velocity, column-velocity*] [, ...])

The first parameter of the CALL SPRITE statement is the sprite number. The sprite number must be an integer from 1 to 28. The sprite number becomes the identifying symbol of the sprite. Each sprite must have its own sprite number. If two sprites are given the same sprite number, only the second sprite will continue to exist.

The second parameter of the CALL SPRITE statement is the ASCII code of the character that is to become the sprite. The sprite will take on the appearance of the character with the corresponding ASCII code.

The third parameter of the CALL SPRITE statement is the color code. The color code specifies the color of the sprite.

The fourth and fifth parameters of the CALL SPRITE statement determine the starting position of the sprite. The fourth parameter is the starting row and the fifth parameter is the starting column. There are 192 visible rows on the screen. As a result, the starting row value can range from 1 to 192. Since there are 256 visible columns on the screen, the column value can range from 1 to 256.

The sixth and seventh parameters of the CALL SPRITE statement determine the speed and direction in which the sprite moves.

The sixth parameter determines the vertical velocity while the seventh parameter determines the horizontal velocity.

The sixth parameter (vertical velocity) can range from -128 to 127. The larger the absolute value of the velocity, the faster the sprite will travel. If a negative velocity is specified, the sprite will move toward the top of the screen. If a positive velocity is specified, the Sprite will move toward the bottom of the screen.

The seventh parameter (horizontal velocity) can range from -128 to 127. The larger the absolute value of the velocity, the faster the sprite will travel. If a negative velocity is specified, the sprite will move from right to left. If a positive velocity is specified, the sprite will move from left to right.

A typical CALL SPRITE statement would appear as follows.

```
CALL SPRITE (#1, 78, 7, 33, 75, 50, -19)
```

Numeric variables, functions and expressions may be used as parameters in a CALL SPRITE statement. The following example demonstrates this principle.

```
CALL SPRITE (#X, ASC(A),X ^ 2,3 + Y,10+X,50,30)
```

Velocity of A Sprite

The velocity of a sprite can be controlled by two different statements, the CALL SPRITE statement and the CALL MOTION statement. The CALL SPRITE statement is used only when creating a sprite. The CALL SPRITE statement establishes the initial velocity of the sprite.

The CALL MOTION statement can be used to change the velocity of a sprite that already exists.

A typical CALL MOTION statement would appear as follows.

```
CALL MOTION (#7, 40, 70)
```

The previous CALL MOTION statement would change the velocity of sprite number seven to a vertical velocity of 40 and a horizontal velocity of 70.

Numeric variables, functions and expressions may also be used as parameters of the CALL MOTION statement, as demonstrated in the following example statement.

```
CALL MOTION (#7, A, A + 30)
```

Position of A Sprite

Three different statements can be used to control and monitor the position of a sprite. These three statements are the CALL SPRITE statement, the CALL POSITION statement, and the CALL LOCATE statement.

The CALL SPRITE statement can only be used when creating a new sprite. The initial position of a sprite is specified in the CALL SPRITE statement.

The CALL POSITION statement can be used to determine the position of a sprite. A typical CALL POSITION statement is as follows.

```
CALL POSITION (#14, X, Y)
```

The previous CALL POSITION statement would assign the row position of sprite number 14 to the variable X, and the column position to the variable Y.

The CALL LOCATE statement can be used to change the position of a sprite. A typical CALL LOCATE statement would appear as follows.

```
CALL LOCATE (#14, 70, 80)
```

The previous CALL LOCATE statement would change the position of sprite number 14 to row 70 and column 80.

Numeric variables, functions and expressions may be used as parameters in both of the preceding statements.

Color of A Sprite

Two different statements can be used to set the color of a sprite. The CALL SPRITE statement and the CALL COLOR statement can both be used to set the color of a sprite. The CALL SPRITE statement can only be used to set the initial color of a sprite.

The color of a sprite can be changed at any time with a CALL COLOR statement. A typical CALL COLOR statement is as follows.

```
CALL COLOR (#5, 9)
```

The previous CALL COLOR statement would change the color of sprite number 5 to red (color code 9).

Numeric variables, functions and expressions may be used as parameters in the CALL COLOR statement.

Causing A Sprite to Disappear

The CALL DELSPRITE statement may be used to cause a specified sprite to disappear. Once a sprite has been deleted using the CALL DELSPRITE statement, the sprite no longer exists.

The following statement is a typical example of a CALL DELSPRITE statement.

```
200 CALL DELSPRITE (#7)
```

The preceding statement would delete sprite number 7.

Numeric variables, functions and expressions may also be used in the CALL DELSPRITE statement. For example, the statement CALL DELSPRITE (#C) is perfectly acceptable as long as the variable C has a value that corresponds to an existing sprite.

The keyword ALL may be used as a parameter in a CALL DELSPRITE statement. In this case, all existing sprites are deleted.

The following statement will cause all sprites that had been defined, to be deleted.

```
CALL DELSPRITE(ALL)
```

Extended BASIC Demonstration Program (1)

The following program incorporates several of the Extended BASIC statements already covered.

```
100 CALL CLEAR
110 CALL HCHAR (12, 14, 43)
120 CALL HCHAR (12, 18, 61)
130 FOR T = 1 TO 2
140 A(T) = INT (RND*5 + 48)
150 CALL SPRITE (#T, A(T), 5, 1, 56 + 32*T, 7, 0)
160 CALL POSITION (#T, X, Y)
170 IF X > 88 THEN CALL MOTION (#T, 0, 0) ELSE 160
180 NEXT T
190 ANS = VAL (CHR$ (A(1))) + VAL (CHR$(A(2)))
200 FOR T = 1 TO 300 :: NEXT T
210 CALL SPRITE (#6, ASC(STR$(ANS)), 5, 89, 155)
220 FOR T =1 TO 700 :: NEXT T
230 CALL DELSPRITE(ALL)
240 GOTO 130
```

The preceding example program consists of a graphics display that performs a series of simple addition problems. Lines 110 and 120 cause the addition and equal signs to appear on the display. Line 130 initiates a FOR/NEXT loop that is repeated twice. Each time the loop is repeated a random number from 0 through 4 is generated at line 140. At line 150, a sprite is defined with the shape of the number. Each sprite is set into motion at the top of the screen and moves slowly toward the bottom.

The statements at line 160 and 170 monitor the position of the sprite, and stop its motion when it reaches the level of the addition symbol. When the sprites are located on both sides of the addition sign, a third sprite is generated at line 210. The value of the third sprite is the sum of the values of the first two sprites. As a result, typical output of the sample program would appear as follows.

2 + 3 = 5

The sample program will automatically be repeated until the CLEAR command (FCTN 4) is executed.

Animation with Sprites

An animation effect can be achieved by using the CALL PATTERN statement to repeatedly change the appearance of a sprite.

The following example program demonstrates how an animation effect can be achieved. The program defines a pair of characters that differ slightly. As the program proceeds, the sprite has the appearance of a face with changing expressions.

```
100 CALL CLEAR :: CALL MAGNIFY(2)
110 CALL CHAR (65, "7E81A58181BD817E")
120 CALL CHAR (66, "7E818181A599817E")
130 CALL SPRITE (#1, 65, 5, 99, 1, 0, 9)
140 CALL PATTERN (#1, 65)
150 FOR A = 1 TO 300 :: NEXT A
160 CALL PATTERN (#1, 66)
170 FOR A = 1 TO 30 :: NEXT A
180 GOTO 140
```

The CALL PATTERN statement can be used to change the appearance of a specified sprite. The CALL PATTERN statements in the preceding program repeatedly changes the appearance of sprite #1 from one shape to another.

The first parameter of the CALL PATTERN statement is the sprite number of the sprite to be changed. The second parameter is the ASCII code of the character that should become the sprite.

Determining the Distance Between Two Sprites

The CALL DISTANCE statement can be used for two different tasks. It can be used to determine the distance between two specified sprites, or the distance between a sprite and a specified location on the screen.

The format of a CALL DISTANCE statement is as follows.

CALL DISTANCE(#*Sprite* { *Sprite* / *row-pixels, column-pixels* } , *numeric variable*)

The value returned in the numeric variable is the square of the distance between the specified sprites or the sprite and the specified position.

The following two examples are typical CALL DISTANCE statements. The first statement determines the distance between two sprites. The second statement determines the distance between a sprite and a location on the screen.

```
CALL DISTANCE (#1, #2, X)
CALL DISTANCE (#1, 77, 39, X)
```

Numeric variables, functions and numeric expressions may be used as parameters in a CALL DISTANCE statement.

Enlarging Sprites

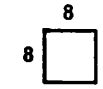
Sprites can be enlarged up to 16 times their original size. The normal size of a sprite is one character (8 pixels x 8 pixels). The largest size a sprite can assume is 16 characters (32 pixels x 32 pixels). This increase in size can be accomplished by a CALL MAGNIFY statement.

The CALL MAGNIFY statement has only one parameter. This parameter determines the size of all the sprites in use.

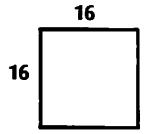
The parameter used in the CALL MAGNIFY statement must be an integer from 1 to 4. Illustration 4-7 shows the effects of the four possible parameters.

Illustration 4-7. MAGNIFICATION PARAMETERS

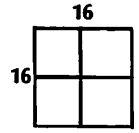
1. Normal size (1 character)



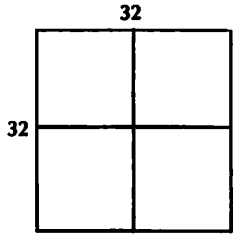
2. 4 times normal size (1 character)



3. 4 times normal size (4 charcters)



4. 16 times normal size (4 characters)



The parameters 2 and 3 seem to have the same effect. However, this is not the case. When a parameter of 1 or 2 is used, only one character is used to define the sprite. When a parameter of 3 or 4 is used, four characters are used to define the sprite. For example, using the ASCII code 92 when creating a sprite, and using a magnification parameter of 3 would cause the sprite to consist of characters 92, 93, 94, and 95.

The characters are grouped in sets of four. When a magnification parameter of 3 or 4 is used, all four characters in the set that contains the specified character are used. Table 4-5 contains a list of the characters that belong to each set. The way in which characters are grouped cannot be changed.

Table 4-5. CHARACTER GROUPS

SET# 1	32	33	34	35
SET# 2	36	37	38	39
SET# 3	40	41	42	43
SET# 4	44	45	46	47
SET# 5	48	49	50	51
SET# 6	52	53	54	55
SET# 7	56	57	58	59
SET# 8	60	61	62	63
SET# 9	64	65	66	67
SET#10	68	69	70	71
SET#11	72	73	74	75
SET#12	76	77	78	79
SET#13	80	81	82	83
SET#14	84	85	86	87
SET#15	88	89	90	91
SET#16	92	93	94	95
SET#17	96	97	98	99
SET#18	100	101	102	103
SET#19	104	105	106	107
SET#20	108	109	110	111
SET#21	112	113	114	115
SET#22	116	117	118	119
SET#23	120	121	122	123
SET#24	124	125	126	127
SET#25	128	129	130	131
SET#26	132	133	134	135
SET#27	136	137	138	139
SET#28	140	141	142	143

The following example demonstrates the effect of using a magnification parameter of 4. Even though character number 78 is used to define the sprite, the sprite actually consists of characters 76, 77, 78, and 79.

```
100 CALL CLEAR :: CALL MAGNIFY(4)
110 CALL SPRITE(#1, 78, 5, 99, 125)
120 GOTO 120
```

Contact Between Sprites

A CALL COINC statement can be used to determine whether or not two sprites have come into contact. The statement can also be used to determine if a sprite is located at a specified position. A typical CALL COINC statement is as follows.

```
CALL COINC(#1, #2, 10, A)
```

The first two parameters are sprite numbers. The third parameter specifies a number of pixels. If the two sprites specified come within the specified number of pixels of each other, a value of -1 is assigned to the numeric variable. Otherwise, the variable is assigned the value 0.

A CALL COINC statement may also specify a sprite and a position. In this case, if the sprite is located at the specified position, the variable will be assigned the value -1. The following is a typical CALL COINC statement that specifies a sprite and a position.

```
CALL COINC (#1, 70, 50, 10, A)
```

The previous statement will report any contact between sprite number 1 and the position 70, 50 (row 70, column 50).

The keyword ALL can also be used as a parameter of the CALL COINC statement. In this case, contact between any two sprites will return a value of -1. The following statement is a CALL COINC statement that uses the ALL parameter.

```
CALL COINC (ALL, A)
```

Extended BASIC Demonstration Program (2)

The following program incorporates several of the Extended BASIC statement including CALL COINC and CALL MAGNIFY.

```
100 CALL CLEAR :: CALL MAGNIFY (2)
110 CALL CHAR (65, "10387CFEFE828282")
120 CALL CHAR (66, "0000003C24FFDB24")
130 CALL SPRITE(#1, 65, 5, 99, 50)
140 CALL SPRITE(#2, 66, 5, 99, 30, 0, -7)
150 CALL COINC (ALL, A)
160 IF A = -1 THEN CALL DELSPRITE(#2) ELSE 150
170 CALL CHAR(65, "10387 C FEFEFEFEFE")
180 FOR T = 1 TO 300 :: NEXT T :: GOTO 110
```

CHAPTER 5.
TI-99/4A BASIC REFERENCE GUIDE

This chapter provides descriptions and examples of the correct syntax for TI Standard BASIC and TI Extended BASIC. Each of the commands, statements, and functions are listed in alphabetical order along with an appropriate abbreviation if applicable.

This manual refers to the BASIC which is built into the TI-99/4A as Standard BASIC. The version of BASIC available in the TI EXTENDED BASIC COMMAND MODULE is refered to as Extended BASIC.

The following notation will be used to describe the configuration of each of the commands, statements, or functions.

1. Capitalized words are keywords.
2. Italicized items are parameters.
3. Items enclosed in brackets [ ] are optional.
4. Ellipsis (...) represent repetition.
5. Punctuation (except brackets and braces) must be included as shown.
6. The following symbols will be used:

ln	Line number
>	Precedes lines typed by user.
□ EX BASIC □ ST BASIC	Indicates whether a statement, command or function may be used in Extended BASIC, Standrd BASIC or both.
A$, B$	String variables
A, B	Numeric variables

ABS

■ EX BASIC
■ ST BASIC

The ABS function returns the absolute value of its argument.

Configuration

X = ABS(*a*)

Example

```
> PRINT ABS(-81)
  81
```

ACCEPT

■ EX BASIC
□ ST BASIC

An ACCEPT statement is used to accept data from the keyboard and assign the data to a variable. An ACCEPT statement is similar in principle to an INPUT statement. However, an ACCEPT statement has more optional features than an INPUT statement.

Configuration

ACCEPT [[AT (*row, column*)] [VALIDATE (*character-type*)] [BEEP] [ERASE ALL] [SIZE (*parameter*)]:] *variable*

When an ACCEPT statement is encountered, the execution of the program does not continue until data is entered at the keyboard.

The AT option allows the data to appear at any location on the display while it is being input. The data will appear at the specified row and column. The display is divided into 24 rows and 28 columns. As a result, the following statement,

ACCEPT AT(12,14):X

will cause the data to appear near the center of the display. When the ENTER key is pressed, the data that appears at the specified location on the display will be assigned to the variable X.

The VALIDATE option is used to restrict the type of data that can be input. The *character type* parameter is used to select the acceptable types of data. Table 5-1 summarizes the effects of the *character type* parameters.

Table 5-1. Character Type Parameters

UALPHA	Only upper case letters are acceptable input.
DIGIT	The digits 0-9 are the only acceptable input.
NUMERIC	Restricts allowable input to integers, floating point numbers, and scientific notation.
" "	Allows only the characters that are included in the quotation marks.

The BEEP option causes a short tone to be generated when the program is ready to accept input.

The ERASE ALL option causes the entire display to be cleared before the data being input appears on the screen.

The SIZE option is used to set a limit to the number of characters that may be accepted as input.

If the SIZE option is not used, characters may be entered until the end of the line is reached.

If the SIZE option includes a positive argument, the specified number of spaces will be cleared before the data is displayed on the screen. The number of characters in the data being entered cannot exceed the specified limit.

The SIZE option may include a minus sign with its argument. The minus sign is used to indicate that the specified number of spaces should not be cleared when the ACCEPT statement is executed. For example, a SIZE(-10) statement sets a maximum data length of 10 characters, but does not clear 10 spaces on the display.

When a SIZE statement includes a minus sign, the data that is entered will take the place of the characters that already appear on the display. However, the data that originally appears on the display will be assigned to the specified variable unless new data is specified. This concept is demonstrated in the following example.

Example

```
>100 FOR A = 0 TO 2
>200 PRINT "**********"
>300 NEXT A
>400 ACCEPT AT(22,2) VALIDATE(UALPHA) SIZE(8):A$
>500 ACCEPT AT(23,2) VALIDATE(UALPHA) SIZE(-8):B$
>600 PRINT::PRINT A$,B$
>RUN
 **********
 *APPLES   *
 *APPLES***
 APPLES              APPLES**
```

The preceding example contains a program that includes two ACCEPT statements. The FOR/NEXT loop at lines 100 through 300 causes three rows of asterisks to be displayed.

The ACCEPT statement at line 400 causes the data being input to appear at location (22,2) on the display. The VALIDATE option indicates that upper case letters are the only type of acceptable data. The SIZE option causes 8 spaces to be cleared before the data is input. As a result, the following display should appear at the bottom left corner of the screen.

```
**********
*        *
**********
```

The data that is input is limited to the 8 blank spaces in the center of the asterisks. In the example, the word APPLE is the data that is entered. When the ENTER key is pressed, the variable A$ is assigned the string value APPLE.

The ACCEPT statement at line 500 is similar to the statement at line 400. However, the data being input is displayed at location (23,2). As a result, the data appears at the line below the previous data.

Once again, the VALIDATE option allows only upper case letters to be input.

The SIZE option imposes an 8 character limit on the data being input. However, the minus sign indicates that the display should not be cleared.

As a result, the display at the bottom of the screen should appear as follows:

```
**********
*APPLES  *
**********
```

As the word APPLE is input, the characters take the place of the asterisks that appear in the last line of the display. However, the asterisks that are not replaced with data will be considered part of the input. When the ENTER key is pressed, the data will be assigned to the variable B$.

The PRINT statements at line 600 display the values of the variables A$ and B$. Notice that the value of B$ contains two asterisks, even though the asterisks were not entered as data.

The following example statements demonstrate the correct format for ACCEPT statements.

Examples

```
ACCEPT AT(5,14) BEEP:A$
ACCEPT AT(1,1) VALIDATE(DIGIT):A
ACCEPT BEEP VALIDATE("YN") SIZE(1):A$
ACCEPT BEEP:A
```

AND

■ EX BASIC
□ ST BASIC

AND is used between two expressions as either a numeric or logical operator.

Configuration

ex AND *ex*

The conditions of true and false are represented in the computer by the logical values -1 and 0. As a result, the logical operators (AND, OR, NOT, and XOR) operate with the logical values -1 and 0. The AND operation can be explained by the following truth table.

ex1	ex2	RESULT
-1	-1	-1
-1	0	0
0	-1	0
0	0	0

AND is generally used in an IF/THEN statement with relational expressions. For example:

```
>10 X = 10
>20 Y = 30
>30 IF X = 10 AND Y >100 THEN GOTO 999
>40 PRINT "CONDITIONS WERE NOT MET"
>999 END
>RUN
 CONDITIONS WERE NOT MET
```

In this example, AND is used in an IF/THEN statement that ends the program if both conditions are true. The first expression of the AND statement is X = 10. This is true because X is assigned the value 10 in line 10. The second expression, Y > 100, is false because Y is assigned the value 30 in line 20. As a result, *ex1* is true and *ex2* is false. This corresponds to the truth table where *ex1=–1* and *ex2=0*. The result from the table is 0 (false), so the condition of the IF/THEN statement is false, and the next line is executed.

When AND is used as a numeric operator, the arguments are rounded off and converted to their binary equivalents. The value of the AND statement is the result of the AND operation on each bit of the values. For example:

```
      6  (binary 0110)
  AND 10 (binary 1010)
  ------------------
      2  (binary 0010)
```

Since the binary equivalent of 6 is 0110, and the binary equivalent of 10 is 1010, the AND operation for each bit has the result 0010. The decimal value of 0010 is 2.

ASC

- EX BASIC
- ST BASIC

The ASC function returns the ASCII code for the first character of its string expression argument. The argument of ASC can either be a string constant or a string variable.

Configuration

X = ASC(*A$*)

Example

```
>PRINT ASC("JONES")
 74
```

ATN

■ EX BASIC
■ ST BASIC

The ATN function returns the arctangent of its argument. The result will be in radians.

Configuration

X = ATN(*A*)

Example

```
>PRINT ATN(.576)
 .5225854816
```

BREAK

■ EX BASIC
■ ST BASIC

A BREAK statement causes the program execution to stop. Line numbers may be included as parameters with the BREAK statement to instruct the computer where to stop the program.

Configuration

BREAK [*ln, ln* . . .]

BREAK can be used in the immediate mode as well as in the program mode. When used in the immediate mode, the BREAK statement must be followed by one or more line numbers. These line numbers instruct the computer where to stop execution of the program. The program will be halted before the specified line is executed.

Example

```
>100 PRINT "JOHN DOE"
>200 PRINT "4444 BAKER ST"
>300 PRINT "CLEVE. OHIO"
```

```
>BREAK 200
>RUN
 JOHN DOE
 * BREAKPOINT IN 200
>RUN
 JOHN DOE
 4444 BAKER ST
 CLEVE. OHIO
```

In the preceding example, BREAK 200 causes the program execution to stop before line 200 is executed. Since BREAK 200 was used in the immediate mode, it only effects the program once. Note when the program is run a second time, there is no break.

The computer can be instructed to continue execution of a program after a break is encountered by using the CON command. In the preceding example, if CON had been entered after the breakpoint instead of RUN, the output would have been as follows:

```
4444 BAKER ST
CLEVE. OHIO
```

When BREAK is used as a statement in a program, line numbers may or may not be included as parameters. If line numbers are not included, program execution will stop at the point where the BREAK appears in the program.

Example

```
>100 PRINT "JOHN DOE"
>200 BREAK
>300 PRINT "4444 BAKER ST"
>400 PRINT "CLEVE. OHIO"
>RUN
 JOHN DOE
 * BREAKPOINT IN 200
>CON
 4444 BAKER ST
 CLEVE. OHIO
```

In this example, the program will stop at line 200 each time it is executed.

In Extended BASIC, multiple statement lines may be used. However, if BREAK is used in a multiple statement line, it must be the last statement in the line. If BREAK is not the last statement, a syntax error will occur when an attempt is made to continue the program.

BYE

■ EX BASIC
■ ST BASIC

The BYE command causes the computer to exit TI BASIC or TI Extended BASIC and return to the master title screen. The master title screen is the display that appears when the computer is powered on.

The QUIT command also returns the computer to the master title screen. However, QUIT will not close any files that had been opened. This could result in the loss of data that has been stored on disk or cassette. The BYE command will close all files that had been opened. Therefore, BYE should generally be used instead of QUIT.

Configuration

BYE

Example

```
>BYE
```

CALL

■ EX BASIC
■ ST BASIC

CALL is used to access subprograms that are stored in the computers memory. Many special purpose subprograms are permanently stored in the computer's memory. However, in Extended BASIC, a SUB statement may be used to define custom made subprograms.

CALL is used most frequently for color, sound, and graphics. The subprograms that are stored in memory are as follows:

CHAR	HCHAR	*PATTERN
*CHARPAT	*INIT	*PEEK
*CHARSET	JOYST	*POSITION
CLEAR	KEY	*SAY
*COINC	*LINK	SCREEN
COLOR	*LOAD	SOUND
*DELSPRITE	*LOCATE	*SPGET
*DISTANCE	*MAGNIFY	*SPRITE
*ERR	*MOTION	VCHAR
GCHAR		*VERSION

*Only in Extended BASIC.

Configuration

CALL *subprogram name* [(*parameters*)]

Example

```
100 CALL CLEAR
```

CALL CHAR

■ EX BASIC
■ ST BASIC

The CALL CHAR statement accesses the subprogram CHAR. The CHAR subprogram allows the user to create graphic characters.

Configuration (EX & ST BASIC)

CALL CHAR (*ASCII code, "string expression"*)

The ASCII code is a widely used system that uses numbers to refer to the characters instead of the characters themselves. Each character defined in the computer is represented by a specific ASCII code. By specifying a particular ASCII code in the CALL CHAR statement, the character that is normally associated with the specified ASCII code can be redefined. For example, the

ASCII code of the character "A" is 65. Therefore, if 65 is used as the parameter in a CALL CHAR statement, the letter "A" will be redefined to some other character.

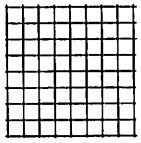
In Standard BASIC, the ASCII codes that can be redefined are codes 32 through 159 inclusive. In Extended BASIC, the ASCII codes that can be redefined are codes 32 through 143 inclusive.

The second parameter of the CALL CHAR statement is the *string expression*. The *string expression* consists of a hexadecimal code. The hexadecimal code is used to define the shapes of characters.

Each character consists of 65 elements or pixels. These 65 pixels form an 8x8 grid the size of one character.

Illustration 5-1 depicts the 8x8 grid that is used to define characters. Characters can be defined by selectively illuminating some of these pixels. The 16 character *string expression* is used to determine the pixels to be illuminated.

Illustration 5-1. Character Grid



In order to use the hexadecimal code, the grid in Illustration 5-1 must be divided into 16 sections of 4 pixels each.

Illustration 5-2 shows an 8x8 grid that has been divided in this manner.

Illustration 5-2. Grid Division

1	□□□□	□□□□	2
3	□□□□	□□□□	4
5	□□□□	□□□□	6
7	□□□□	□□□□	8
9	□□□□	□□□□	10
11	□□□□	□□□□	12
13	□□□□	□□□□	14
15	□□□□	□□□□	16

Each one of these sections can be defined with one character of the hexadecimal code. Any combination of the 4 pixels can be illuminated. Illustration 5-3 shows the hexadecimal codes that correspond to the 16 possible combinations.

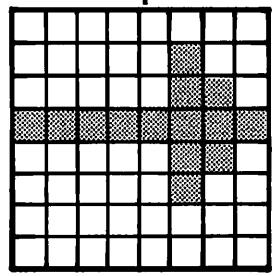
Illustration 5-3. Hexadecimal Code

PIXELS	HEXADECIMAL CODE
□□□□	0
□□□■	1
□□■□	2
□□■■	3
□■□□	4
□■□■	5
□■■□	6
□■■■	7
■□□□	8
■□□■	9
■□■□	A
■□■■	B
■■□□	C
■■□■	D
■■■□	E
■■■■	F

The 8x8 grid that makes up one character can be described in 16 hexadecimal characters. These 16 hexadecimal characters make up the *string expression* used in the CALL CHAR statement.

The first character in the *string expression* will describe section 1 of the 8x8 grid. The second character will describe section 2, and so on.

Example



0	0
0	4
0	6
F	F
0	6
0	4
0	0
0	0

```
100 CALL CLEAR
200 CALL CHAR(143,"000406FF06040000")
300 CALL HCHAR(12,16,143)
```

The preceding example shows how a CALL CHAR statement is used to create a character. The first step in creating a character is to draw the character within an 8x8 grid. Then, translate the drawing into a hexadecimal code using Illustration 5-3.

The hexadecimal code for the specified character can now be used in a CALL CHAR statement.

The program in the preceding example will place a small arrow in the middle of the screen. Line 100 will clear the screen. Line 200 will use the string expression to define character number 143.

Line 300 will display the character in the middle of the screen.

Any type of character can be defined using the CALL CHAR statement. The following example contains a program that uses the CALL CHAR statement to define the character π (PI).

Example

```
>100 CALL CHAR (130,"007E242424244600")
>200 B = 3.141592654
>300 PRINT CHR$(130); "=";B
>RUN
 π = 3.141592654
```

The *string expression* in the CALL CHAR statement must be 16 characters in length. If less than 16 characters are used, the missing characters are assumed to be zeros. In Standard BASIC, if more than 16 characters are used, the string will be truncated. In Extended BASIC, however, specifying more than 16 characters causes the character with the next highest ASCII code to be redefined.

Configuration (EX BASIC)

CALL CHAR (*ASCII code, "string expression"* [,...])

Extended BASIC allows many characters to be defined in one statement. The *string expression* may be as long as 64 characters. The first 16 characters will redefine the character with the specified ASCII code. Characters 17 through 32 will redefine the character with the next higher ASCII code and so on.

More than one character can be redefined by listing ASCII codes and hexadecimal codes one after the other in the CALL CHAR statement.

Examples

```
CALL CHAR(66,"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF")
CALL CHAR(66,"FFFFFFFFFFFFFFFF",67,"FFFFFFFFFFFFFFFF")
```

The previous example contains two statements that have the same results. Both statements redefine the characters 66 and 67 as solid blocks.

CALL CHARPAT

■ EX BASIC
□ ST BASIC

The CALL CHARPAT statement accesses the subprogram CHARPAT. The CHARPAT subprogram returns the 16 character hexadecimal code that is used to create a particular character.

Configuration

CALL CHARPAT (*ASCII-code*,A$[,...])

The first parameter of the CALL CHARPAT statement is the ASCII code of the character whose hexadecimal code will be returned.

A$ is a string variable that will be assigned the hexadecimal code of the specified character.

Example

```
>CALL CHARPAT(65,A$)
>PRINT A$
 003844447C444444
```

In the preceding example, the CALL CHARPAT statement is used to return the hexadecimal code of the character A (ASCII code 65). This hexadecimal code is assigned to the variable A$. A PRINT statement is used to display the value of the variable A$.

CALL CHARSET

■ EX BASIC
□ ST BASIC

The CALL CHARSET statement accesses the subprogram CHARSET. The CHARSET subprogram causes all characters in the character set to return to their standard character definition.

Configuration

CALL CHARSET

The CHARSET subprogram will also reset character colors to their standard colors.

Example

```
100 CALL CHARSET
```

CALL CLEAR

■ EX BASIC
■ ST BASIC

CALL CLEAR accesses the subprogram CLEAR. CLEAR can be used in either the immediate mode or the program mode to clear the entire screen.

Configuration

CALL CLEAR

Example

```
>CALL CLEAR
```

CALL COINC

■ EX BASIC
□ ST BASIC

The CALL COINC statement accesses the subprogram COINC. The COINC subprogram determines if there has been a collision between two sprites. The COINC subprogram can also determine if a sprite passes over a specified position on the screen. The COINC subprogram returns a true or false value depending on whether or not the specified condition has been met.

Configuration

$$\text{CALL COINC}\left(\left\{\begin{array}{l}\text{ALL}\\ \#sprite,\ \#sprite,\ int\\ \#sprite,\ row\text{-}pixel,\ column\text{-}pixel,\ range\end{array}\right\},var\right)$$

The last parameter in the CALL COINC statement must be a numeric variable. The value assigned to this numeric variable will be either -1 or 0. A value of -1 will be assigned to the variable if the specified collision has occurred. A value of 0 will be assigned to the variable if the specified collision has not occurred.

If the keyword ALL is used in the CALL COINC statement, a collision between any two sprites causes a value of -1 to be assigned to the variable.

If two sprite numbers are specified, a collision between the two sprites will cause a value of -1 to be assigned to the variable.

When a sprite and a position are specified, a value of -1 is assigned to the variable if the specified sprite passes over the specified position.

Positions are specified by *row-pixel* and *column-pixel*. Pixels are very small dots which make up the display on the screen. There are 256 column-pixels and 193 row-pixels that are visible on the screen. Therefore, the *row-pixel* in the CALL COINC statement is a number from 1 to 193 and the *column-pixel* is a number from 1 to 256.

When two sprites or a sprite and a position are specified, a *range* must also be specified. The *range* specifies how close two sprites or a sprite and a position must be before it is considered a collision. The *range* is an integer that represents a number of pixels.

A numeric variable (*var*) must always be used with CALL COINC. The variable is assigned the value of -1 for true and 0 for false.

Example

```
100 CALL CLEAR :: CALL MAGNIFY(2)
200 CALL SPRITE(#1,49,5,1,1,34,56)
300 CALL SPRITE(#2,50,5,193,256,-34,-56)
400 CALL COINC(#1,#2,10,A)
500 IF A = -1 THEN CALL DELSPRITE(#1) :: CALL
PATTERN(#2,51)
600 GOTO 400
```

The preceding example contains a program that uses a CALL COINC statement to determine whether or not two sprites have collided. Line 100 clears the screen and sets a magnification factor of two. Line 200 and 300 create two sprites that have the appearances of the numbers "1" and "2". Line 400 assigns A the value of -1 or 0 depending on whether or not the two sprites have collided. Line 500 is an IF/THEN statement that deletes sprite number 1 and changes the appearance of sprite number 2 if a collision occurs. Line 600 branches program control back to line 400.

When the program is executed, two sprites in the shapes of the numbers "1" and "2" begin moving along the screen. When the two sprites collide (within three pixels) sprite number one is deleted and sprite number two takes on the appearance of the number "3".

CALL COLOR

- EX BASIC
- ST BASIC

The CALL COLOR statement accesses the subprogram COLOR. The COLOR subprogram is used to change the color of characters.

Configuration (EX & ST BASIC)

CALL COLOR (*set#, foreground, background*)

In Standard BASIC, the ASCII codes (32 through 159) are divided into 16 sets. In Extended BASIC, the ASCII codes (32 through 143) are divided into 14 sets.

The *set#* used in the CALL COLOR statement must be an integer from 1 to 16. The *set#* specifies the characters to be colored. The ASCII codes and corresponding set numbers can be found in Table 5-2.

For example, if *set#* 8 is specified in a CALL COLOR statement, all characters having ASCII codes from 88 to 95 will be colored as specified in the second and third parameters.

The *foreground* and *background* colors are specified by the integers 1 through 16. Each number specifies a particular color, as indicated in Table 5-3.

Table 5-2. Color Sets

set #	ASCII codes	set #	ASCII codes
1	32-39	9	96-103
2	40-47	10	104-111
3	48-55	11	112-119
4	56-63	12	120-127
5	64-71	13	128-135
6	72-79	14	136-143
7	80-87	15	144-151
8	88-95	16	152-159

Table 5-3. Color Codes

Color Code	Color	Color Code	Color
1	Transparent	9	Medium Red
2	Black	10	Light Red
3	Medium Green	11	Dark Yellow
4	Light Green	12	Light Yellow
5	Dark Blue	13	Dark Green
6	Light Blue	14	Magenta
7	Dark Red	15	Gray
8	Cyan	16	White

The *foreground* color specifies the color of the character. The *background* color specifies the color of the screen behind the character.

Example

```
100 CALL CLEAR
200 CALL COLOR(6,2,5)
300 PRINT "ABCDEFGHIJKLMNOPQRST"
400 GOTO 400
```

The previous example contains a program that uses the CALL COLOR statement.

Line 100 will clear the screen. Line 200 will cause all characters having ASCII codes 72 through 79 to have a black foreground and blue background. Line 300 will display the characters A through T. Line 400 will create a loop that keeps the program from ending.

When the program in the previous example is executed, the letters A through T are displayed on the screen. However, only letters H through O are colored. The letters H through O are the only characters in set number 6. Therefore, they are the only characters specified in the CALL COLOR statement.

When the preceding program is stoped, all characters will return to their standard colors. The CALL COLOR statement only has an effect while the program is being executed.

Example

```
100 CALL CLEAR
200 CALL CHAR(65,"007E7E7E7E7E7E00")
300 CALL HCHAR(1,1,65,768)
400 CALL COLOR(5,7,5)
500 GOTO 500
```

The preceding example contains a program that uses CALL COLOR. Line 100 of the program will clear the screen. Line 200 is a CALL CHAR statement that defines a small square. Line 300 will fill the screen with this small square. Line 400 will cause the color of the square to be red and the background color to be blue. Line 500 causes a loop that prevents the program from ending.

To stop the program, press FCTN 4 (CLEAR).

Configuration (EX BASIC)

CALL COLOR(#*sprite*, *foreground*[,...])

In Extended BASIC, a CALL COLOR statement can be used to change the color of any sprite.

The *sprite* is an integer between 1 and 28 inclusive that specifies the sprite to be colored. The *foreground* is an integer between 1 and 16 inclusive that specifies the color of the sprite.

Example

```
100 CALL COLOR(#3,5)
```

The statement in the previous example would change the color of sprite number three to blue.

CALL DELSPRITE

■ EX BASIC
□ ST BASIC

The CALL DELSPRITE statement accesses the subprogram DELSPRITE. The DELSPRITE subprogram is used to delete sprites. Once a sprite is deleted, it cannot be used again unless it is redefined with a CALL SPRITE statement.

Configuration

CALL DELSPRITE ({ ALL / #*sprite* [#,*sprite*...] })

If the keyword ALL is used with DELSPRITE, all sprites will be deleted. If one or more sprite numbers are specified, only the specified sprites will be deleted.

Example

```
100 CALL DELSPRITE(#1,#7)
```

CALL DISTANCE

■ EX BASIC
□ ST BASIC

The CALL DISTANCE statement accesses the subprogram DISTANCE. The DISTANCE subprogram is used to determine the square of the distance between two sprites. The DISTANCE subprogram can also be used to determine the square of the distance between a sprite and a position on the screen.

Configuration

CALL DISTANCE (#*sprite*, { #*sprite* / *row-pixel*, *column-pixel* }, *variable*)

The parameters of CALL DISTANCE must be either two sprites or a sprite and a screen position. In either case, the square of distance is assigned to the specified *variable*. The *variable* must be a numeric variable.

The DISTANCE subprogram determines the square of the approximate number of pixels that separate two objects. The upper left corner of each sprite is the point of reference.

For example, two sprites that are displayed with 1 inch between their upper left corners would cause the DISTANCE subprogram to indicate a value of 625. Since the square root of 625 is 25, the two sprites are about 25 pixels apart.

The largest value that can be returned by the CALL DISTANCE statement is 32767.

Examples

```
CALL DISTANCE (#1,#2,A)
CALL DISTANCE (#1,77,120,B)
```

The first statement in the preceding example will determine the distance between two sprites. This distance will be assigned to the variable A.

The second statement in the preceding example will return the distance between sprite number one and position 77,120. This distance will be assigned to the variable B.

CALL ERR

■ EX BASIC
□ ST BASIC

The CALL ERR statement accesses the subprogram ERR. The ERR subprogram is generally used after an error occurs. This subprogram allows information about the error to be assigned to variables.

Configuration

CALL ERR(A,B[,C,D])

The value assigned to the first variable (A) is the error code that corresponds to the error.

The value assigned to the second variable (B) is generally -1. If the error that occurred was a file error, the value assigned to the second variable will be positive and correspond to the file that caused the error.

The value assigned to the third variable (C) is always 9.

The value assigned to the fourth variable (D) is the line number where the error occurred.

Example

```
100 INPUT A
200 ON ERROR 500
300 PRINT SQR(A)
400 END
500 CALL ERR(B,C)
600 IF B=74 THEN PRINT "SQUARE ROOT OF ";A;" IS UNDEFINED"
```

The previous example contains a program that uses the CALL ERR statement. Line 100 of the program will allow a number to be assigned to the variable A. Line 200 will transfer program control to line 500 if an error occurs. Line 300 will print the square root of the value of the variable A. Line 400 will end the program.

Line 500 and 600 will only be executed if an error occurs. Line 500 will assign the error code to the variable B. Line 600 will print a message if the error code is 74 (BAD ARGUMENT error).

CALL GCHAR

- EX BASIC
- ST BASIC

The CALL GCHAR statement accesses the subprogram GCHAR. The GCHAR subprogram is used to determine the character that is located at a particular position on the screen.

Configuration

CALL GCHAR (*row, column, variable*)

The GCHAR subprogram will return the ASCII code of the character located at the specified *row* and *column*.

The screen is divided into 24 rows and 32 columns. Therefore, any number from 1 to 24 is an acceptable row number and any number from 1 to 32 is an acceptable column number.

The *variable* is a numeric variable that will be assigned the ASCII code of the character in the specified position.

Example

```
100 CALL GCHAR(15,17,A)
```

CALL HCHAR

■ EX BASIC
■ ST BASIC

The CALL HCHAR statement accesses the subprogram HCHAR. The HCHAR subprogram is used to place characters at specified positions on the screen.

Configuration

CALL HCHAR (*row, column, ASCII code* [,*repetitions*])

The first two parameters, *rows* and *column*, determine the position where the character will be displayed. The screen is divided into 24 rows and 32 columns. Therefore, the specified row must be a number from 1 to 24 and the specified column must be a number from 1 to 32.

The third parameter, *ASCII code*, is the ASCII code of the character that is to be placed at the specified position.

The fourth parameter, *repetition*, is optional. *repetition* must be a number from 1 to 32767. The *repetition* specifies the number of times the specified character is to be repeated. Characters will be repeated horizontally the specified number of times.

If no parameter is included for *repetition*, the character will be displayed only once.

Example

```
CALL HCHAR(12,16,42)
CALL HCHAR(2,2,42,12)
```

The preceding example contains two CALL HCHAR statements. The first statement will place an asterisk (*) in the middle of the screen (position 12,16). The second statement will print 12 asterisks (*) starting at the upper left hand corner of the screen (position 2,2).

CALL INIT

■ EX BASIC
□ ST BASIC

The CALL INIT statement accesses the subprogram INIT. The subprogram INIT prepares the computer to run assembly language subprograms. Assembly language programs cannot be executed unless a memory expansion unit is used with the TI-99/4A.

Configuration

CALL INIT

The CALL INIT statement must be entered before any other statements are used to access assembly language subprograms.

A set of assembly language supporting routines are loaded into the memory expansion when the CALL INIT command is entered.

A SYNTAX ERROR will occur if the memory expansion unit is not powered on.

Example

CALL INIT

CALL JOYST

■ EX BASIC
■ ST BASIC

The CALL JOYST statement accesses the subprogram JOYST. The JOYST subprogram is used to determine the position of the joystick controllers.

Configuration

CALL JOYST (*joyst#, variable 1, variable 2*)

The *joyst#* is the number of the joystick to be used. *joyst#* must be an integer from 1 to 4. However, only joysticks 1 and 2 are accessible.

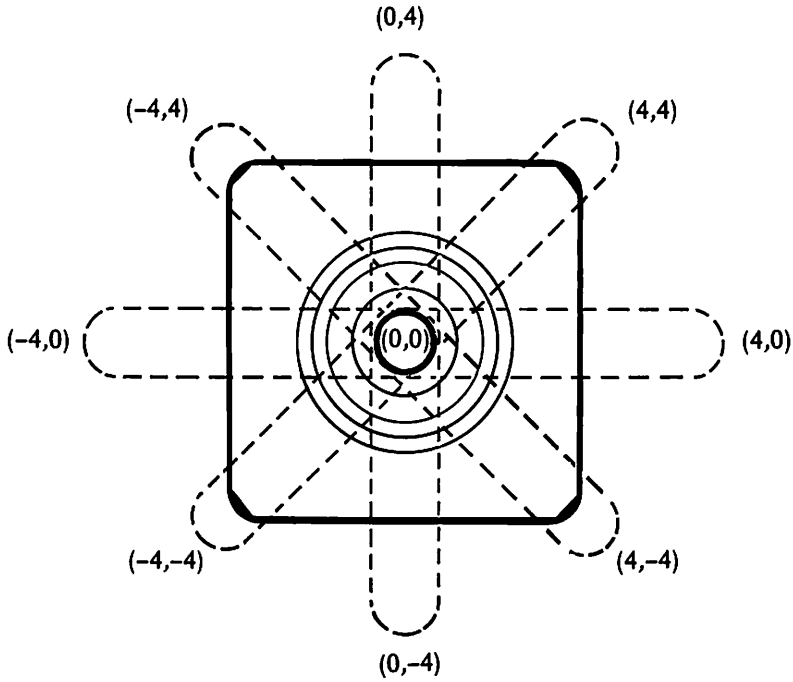
The JOYST subprogram returns two values corresponding to the position of the joystick. These values are assigned to *variable 1* and *variable 2*.

Illustration 5-4 shows the value that will be returned for every joystick position.

Example

```
100 CALL JOYST(1,X,Y)
```

Illustration 5-5 CALL JOYST Joystick Positions



CALL KEY

- EX BASIC
- ST BASIC

CALL KEY accesses the subprogram KEY. KEY is used to accept input from the keyboard while a program is being executed.

Configuration

CALL KEY (*mode, variable 1, variable 2*)

The CALL KEY statement assigns a value to *variable 1* that corresponds to the key that is being pressed on the keyboard. *variable 1* is assigned the value -1 if no key is being pressed.

The *mode* determines the values that are returned by each character. The *mode* must be an integer between 0 and 5 inclusive. Generally, a *mode* of 0 is chosen. A 0 *mode* specifies that all characters return their standard ASCII values. If a *mode* other than 0 is chosen, non-standard values for the characters are returned.

variable 2 is assigned a value of -1, 0 or 1. The value assigned to this variable is determined by the relationship between the last CALL KEY statement and the current CALL KEY statement.

variable 2 is assigned the value -1 if the CALL KEY statement assigns the same character code to *variable 1* in two consecutive statements. *variable 2* is assigned the value 1 only when *variable 1* is assigned a new character code. *variable 2* is assigned the value 0 only when *variable 1* is assigned the value -1. This situation occurs if no keys are being pressed when the statement is executed.

Example

```
100 CALL KEY(0,A,B)
200 IF (B = 0) + (B = -1) THEN 100
300 PRINT CHR$(A)
400 GOTO 100
```

The previous example shows a program that uses the CALL KEY statement to print the character corresponding to the key that was pressed.

Line 100 of the program is a CALL KEY statement. The *mode* is set to 0. Therefore, the standard ASCII values of the characters will be used.

Line 200 branches program control back to line 100 if B = 0 or B = -1. Therefore, if no key is pressed, or the same key is pressed, program control is passed to line 100.

Line 300 prints the character that corresponds to the ASCII code returned in A. Line 400 branches program control to line 100.

To stop the program, press FCTN 4 (CLEAR).

CALL LINK

■ EX BASIC
□ ST BASIC

CALL LINK accesses the subprogram LINK. The LINK subprogram passes program control to an assembly language subprogram.

Configuration

CALL LINK (*subprogram name* [,*variable list*])

The *subprogram name* is the name of the assembly language subprogram. The *variable list* passes parameters from the main program to the assembly language subprogram.

An assembly language subprogram must have been previously loaded into the memory expansion unit before CALL LINK can be used.

CALL LOAD

■ EX BASIC
□ ST BASIC

The CALL LOAD statement is used to store bytes of information in specified memory locations. The CALL LOAD statement can only be used when a memory expansion unit is in use. Also the CALL INIT command must be executed before the CALL LOAD command is used. The CALL LOAD statement can also be used to retrieve an assembly language subprogram from a data file and store it in a memory expansion unit.

Configuration

CALL LOAD (*"filename"* / *memory address, byte 1* [, *byte 2, ...*])

An assembly language subprogram can be loaded into the memory expanison unit by using the CALL LOAD statement along with the filename of the assembly language subprogram. The following example will load the assembly language subprogram named "START" from disk drive 1.

Example

```
>CALL LOAD ("DSK1.START")
```

The CALL LOAD statement can be used to change individual memory addresses. Also, if more than one byte of information is specified, each following byte is stored in each successive memory address.

Example

```
>CALL LOAD (8734,65,72)
```

The previous CALL LOAD statement will store 65 in memory address 8734 and 72 memory address 8735.

The CALL LOAD statement can only store information in memory addresses that are part of RAM. If a CALL LOAD statement specifies a memory location that is part of the ROM, the CALL LOAD statement has no effect.

CALL LOCATE

■ EX BASIC
□ ST BASIC

The CALL LOCATE statement accesses the subprogram LOCATE. The LOCATE subprogram is used to change the position of a sprite.

Configuration

CALL LOCATE (#*sprite*, *row*, *column* [,...])

The *sprite* is an integer from 1 to 28 that indicates the sprite to be moved. The *row* is an integer from 1 to 192. The *column* is an integer from 1 to 256.

Example

```
>100 CALL CLEAR
>200 CALL MAGNIFY(2)
>300 CALL SPRITE(#1,65,5,99,125)
>400 CALL LOCATE(#1,INT(RND*192) +1, INT(RND*256) +1)
>500 GOTO 400
>RUN
```

The preceding example contains a program that will continually relocate a sprite.

Line 100 will cause the display to be cleared. Line 200 will cause all sprites to be magnified. Line 300 will cause a sprite to appear in the shape of the letter A. Line 400 will cause the sprite to be relocated at a random position on the screen. Line 500 will branch program control to line 400.

Press FCTN 4 (CLEAR) to stop the program.

CALL MAGNIFY

■ EX BASIC
□ ST BASIC

CALL MAGNIFY accesses the subprogram MAGNIFY. The subprogram MAGNIFY is used to increase the size of sprites.

Configuration

CALL MAGNIFY (*a*)

A sprite can be magnified in two ways. The first method is to increase the size of the character that makes up the sprite. The second method is to use more than one character to define a sprite.

The parameter in the MAGNIFY subprogram must be an integer from 1 to 4 that determines how a sprite will be magnified.

A parameter of 1 causes all sprites to consist of one character. This character will be of normal size. Therefore, all sprites will be of normal size.

A parameter of 2 causes all sprites to consist of one character. However, this character will be four times its original size. Therefore, all sprites will be four times their original size.

A parameter of 3 causes all sprites to consist of four characters. These four characters will all be normal sized. Therefore, all sprites will be four times their original size.

A parameter of 4 causes all sprites to consist of four characters. However, these characters will be four times their original size. Therefore, all sprites will be sixteen times their original size.

Example

```
100 CALL CLEAR
200 CALL SPRITE(#1,76,6,99,115)
300 ACCEPT AT(23,1) VALIDATE ("1234") SIZE (1):A
400 CALL MAGNIFY(A)
500 GOTO 300
```

The preceding example contains a program that shows the different effects that a CALL MAGNIFY statement can produce.

When executed, the preceding program causes a sprite to appear in the middle of the screen. The sprite has the shape of the letter L.

The cursor then appears at the bottom of the screen. An integer from 1 to 4 may now be entered. This integer is then used in a CALL MAGNIFY statement to magnify the sprite.

Line 100 in the preceding program will clear the screen. Line 200 will create a sprite with the shape of the letter L and place it in the middle of the screen. Line 300 will input a number from 1 to 4 and assign it to the variable A. Line 400 is a CALL MAGNIFY statement that uses the variable A as its parameter. Line 500 will branch program control to line 300.

Press the FCTN 4 (CLEAR) key to stop program execution.

CALL MOTION

■ EX BASIC
□ ST BASIC

The CALL MOTION statement accesses the subprogram MOTION. The MOTION subprogram is used to change the direction and speed of a sprite.

Configuration

CALL MOTION (#*sprite, vertical speed, horizontal speed*)

The CALL MOTION statement can be used to change the *vertical speed* and *horizontal speed* of a sprite.

The *sprite* is an integer from 1 to 28 that indicatees the sprite to be affected. The *vertical speed* is an integer from -128 to 127. A negative *vertical speed* will cause a sprite to move up the screen.

A positive *vertical speed* will cause a sprite to move down the screen.

The *horizontal speed* is an integer from -128 to 127. A negative *horizontal speed* will cause a sprite to move left on the screen. A positive *horizontal speed* will cause a sprite to move right on the screen.

The absolute value of the numbers specified as the second and third parameters determine the speed of the sprite. If the absolute value of the specified number is close to zero, the sprite will move slowly. If the absolute value of the specified number is much greater than zero, the sprite will move quickly.

Example

```
100 CALL CLEAR
200 CALL MAGNIFY(2)
300 CALL SPRITE(#1,65,5,99,115)
400 INPUT A,B
500 CALL MOTION(#1,A,B)
600 GOTO 400
```

The preceding example contains a program that uses a CALL MOTION statement. When the program is executed, a sprite with the shape of the letter A appears in the middle of the screen. The cursor then appears at the bottom of the screen. The sprite can be set into motion in any direction and at any speed by entering two number between -128 and 127.

Line 100 of the preceding program will clear the screen. Line 200 will magnify all sprites. Line 300 will create a sprite with the shape of the letter A and places it in the center of the screen. Line 400 will input two numbers and assign them to the variables A and B. Line 500 is a CALL MOTION statement that uses the variable A and B to change the motion of sprite #1. Line 600 will branch program control to line 400.

■ EX BASIC
□ ST BASIC

CALL PATTERN

CALL PATTERN accesses the subprogram PATTERN. The subprogram PATTERN is used to change the appearance of a sprite without changing any other characteristics of that sprite.

Configuration

CALL PATTERN (#*sprite*, *ASCII code* [,III])

The first parameter of the CALL PATTERN statement specifies the number of the sprite to be changed. The second parameter specifies the ASCII code of the character that will be used in the specified sprite.

Example

```
100 CALL CLEAR :: CALL MAGNIFY(2)
110 CALL CHAR(65,"3C667E3C18244281")
120 CALL CHAR(66,"3C7E7E3C18242424")
130 CALL SPRITE(#1,66,5,100,100,0,12)
140 FOR T = 1 TO 30 :: NEXT T
150 CALL PATTERN(#1,65)
160 FOR T = 1 TO 30 :: NEXT T
170 CALL PATTERN(#1,66)
180 GOTO 140
```

The preceding example contains a program that uses the CALL PATTERN statement to create an animation effect.

Line 100 of the program clears the screen and causes all sprites to be magnified. Line 110 and 120 define two different characters that will be used in creating the animation effect. Line 130 creates a sprite using character 66. Line 140 uses a FOR/NEXT loop to cause a delay. Line 150 causes the sprite number one to become character 65. Line 160 uses a FOR/NEXT loop to create a second delay. Line 170 changes sprite number one back to character 66. Line 180 causes the last five lines of the program to be repeated.

Press CLEAR (FCTN 4) to end the program.

CALL PEEK

■ EX BASIC
□ ST BASIC

The CALL PEEK statement accesses the subprogram PEEK. The subprogram PEEK is used to recover the value in a memory location.

Configuration

CALL PEEK (*memory location, variable list*)

A memory location contains an integer value between 0 and 255. The first parameter of the CALL PEEK statement must be an integer from -32768 to 32767. This number specifies the memory location. A numeric overflow error occurs if the specified memory location is greater than 32767 or less than -32768. If the specified memory location is not an integer, it is rounded off.

The first variable in the *variable list* is assigned the value contained in the specified memory location. The second variable in the *variable list* is assigned the value contained in the next highest memory location and so on.

The CALL PEEK statement is generally used when executing an assembly language subprogram.

CALL POSITION

■ EX BASIC
□ ST BASIC

CALL POSITION accesses the subprogram POSITION. The POSITION subprogram returns the coordinates of the location of a specifies sprite.

Configuration

CALL POSITION (#*sprite, variable 1, variable 2* [,...])

The *sprite* is the number of the sprite whose position is to be returned.

The position of the sprite will be assigned to *variable 1* and *variable 2*. The row pixel position will be assigned to *variable 1*. The column pixel position will be assigned to *variable 2*.

Example

CALL POSITION (#1,A,B)

CALL SAY

■ EX BASIC
□ ST BASIC

The CALL SAY statement accesses the subprogram SAY. The SAY subprogram is used to activate the Solid State Speech Synthesizer. A CALL SAY statement has no effect if the Speech Synthesizer is not properly installed.

Configuration

CALL SAY ("*word string*,"*X$*,...)

The *word string* may include any word, or any combination of words that are contained in the speech synthesizer's resident vocabulary. A listing of these words can be found in Appendix F.

The resident vocabulary also includes phrases. When using a phrase, place a "#" on either end of the phrase.

If a word that is not included in the resident vocabulary is used, the computer will spell that word. If the Speech Synthesizer does not recognize a character in a word string, the word "uh oh" is produced instead.

Example

```
>CALL SAY ("I AM THE #TEXAS INSTRUMENTS#
 HOME COMPUTER")
```

A CALL SPGET statement can be used to generate "speech strings." These strings can be processed more quickly then word strings. When speech strings are used in a CALL SAY statement, they must be separated from word strings by commas.

CALL SAY statements require that word strings and speech strings be used alternately, and that the first value is a word string. As a result, a CALL SAY statement must have a comma or a word string as the first data item.

Example

```
>CALL SPGET ("HELLO",A$)
>CALL SAY ("GOODBYE",A$)
>CALL SAY (,A$)
```

CALL SCREEN

■ EX BASIC
■ ST BASIC

CALL SCREEN accesses the subprogram SCREEN. The SCREEN subprogram is used to change the color of the screen.

Configuration

CALL SCREEN (*a*)

The parameter for CALL SCREEN is an integer from 1 to 16 that specifies the color of the screen. The color codes used in the CALL SCREEN statement are the same as those used in the CALL COLOR statement. Table 5-3 on page 142 lists all the color codes.

Example

CALL SCREEN (9)

CALL SOUND

■ EX BASIC
■ ST BASIC

CALL SOUND accesses the subprogram SOUND. The subprogram is used to generate musical notes and noises.

Configuration

CALL SOUND (*delay, frequency 1, volume 1*[,..., *frequency 4, volume 4*])

The CALL SOUND statement can simultaneously generate up to three tones and one noise.

The *delay* parameter determines the duration of the tone or noise. There can only be one *delay* specified within a CALL SOUND statement. The specified *delay* will affect all of the tones and noises that are specified in the statement.

The *delay* is measured in milliseconds. The range of the delay is from 1 to 4250 and from -1 to -4250.

When a positive *delay* is specified, the specified tone will continue to be generated until the specified duration is complete. When a negative *delay* is specified, the specified tone will be generated until the specified duration is complete or a new CALL SOUND statement is encountered.

Program execution will not stop while a tone is being generated.

The *frequency* for a tone is measured in hertz. The *frequency* may vary from 110 to 44733, where 110 is the lowest possible tone and 44733 is the highest possible tone. Frequencies much higher than 10,000 hertz generally cannot be heard.

The *frequency* for a noise must be an integer from -1 to -8. When used in a CALL SOUND statement, each of these integers will cause a different noise to be generated.

The *volume* is an integer from 0 to 30 that determines the volume of each tone or noise. Using a 0 for the *volume* in a CALL SOUND

statement causes the specified tone or noise to be generated at the highest possible volume. Using a 30 for the *volume* in a CALL SOUND statement causes the specified tone or noise to be generated at the lowest possible volume.

Example

```
100 FOR A = 110 TO 44733
200 CALL SOUND(-100,A,0)
300 NEXT A
```

The program in the previous example will generate all the possible tones. Line 100 initializes a FOR/NEXT loop that is used to specify the frequency. Line 200 is a CALL SOUND statement that uses the value of the variable A as the *frequency*. Line 300 completes the FOR/NEXT loop.

Example

```
100 FOR A = -8 to -1
200 CALL SOUND(4000,A,0)
300 PRINT A
400 NEXT A
```

The preceding example will generate all the possible noises. The number of the noise is displayed when it is generated.

CALL SPGET

■ EX BASIC
□ ST BASIC

CALL SPGET accesses the subprogram SPGET. The SPGET subprogram returns the speech string that corresponds to the specified word string. A speech string is a set of special characters that can be pronounced more quickly than a wordstring.

Configuration

CALL SPGET (*"wordstring", string variable*)

The first parameter is any word that can be found in the resident vocabulary. The second parameter is a string variable. The speech string that corresponds to the specified word string is assigned to the specified string variable. Using this string variable in a CALL SAY statement allows the computer to process speech more quickly than if a wordstring was used in a CALL SAY statement

Example

```
>CALL SPGET ("GOODBYE",A$)
>CALL SAY ("HELLO",A$)
>CALL SAY (,A$)
```

CALL SPRITE

■ EX BASIC
□ ST BASIC

The CALL SPRITE statement accesses the subprogram SPRITE. The SPRITE subprogram is used to create graphic characters that can move smoothly across the screen. These characters are called sprites.

Configuration

CALL SPRITE (*#sprite, char-code, color, row-pixel,*
column-pixel[,vertical speed, horizontal speed][,...])

A sprite can be placed at any position on the screen. A sprite can also be set into motion to move in any direction at a wide range of speeds.

Once set into motion, a sprite will move smoothly across the screen. The motion of a sprite will not change unless the program ends or a program statement changes the motion.

When a sprite and a fixed object meet, the sprite will pass over that object. When two sprites meet, the sprite with the lower number will pass over the sprite with the higher number.

There can be up to 28 sprites on the screen at any one time. However, no more than 4 sprites can appear in the same row. If more than four sprites are positioned in a row, only the four sprites with the lowest numbers will appear.

There are five essential parameters and two optional parameters in a CALL SPRITE statement. The first parameter is the *sprite*. The *sprite* is a number from 1 to 28. This number becomes the identifying number of the sprite. The sprite number is used when referencing a sprite.

The *ASCII code* is a number from 32 to 143 that specifies the character that will be used as the sprite. Each ASCII code corresponds to a predefined character. However, the CALL CHAR statement can be used to redefine the standard ASCII characters. Therefore, a sprite can be given any shape.

The third parameter is *color*. The *color* is an integer between 1 and 16 inclusive. This number specifies the color of the sprite. The color codes are listed in Table 5-3, page 142.

The fourth parameter is the *row-pixel*. The *row-pixel* specifies the starting row position of the sprite. The screen is divided into 192 row pixels. Therefore, *row-pixel* must be a number between 1 and 192 inclusive.

The fifth parameter is the *column-pixel*. The *column-pixel* specifies the starting column position of the sprite. The screen is divided into 256 column pixels. Therefore, *column-pixel* must be a number between 1 and 256 inclusive.

The two optional parameters, *vertical speed* and *horizontal speed*, are used when the sprite is to be set in motion.

The *vertical speed* is in integer from -128 to 127. A positive *vertical speed* will cause a sprite to move toward the top of the screen. A negative *vertical speed* will cause a sprite to move toward the bottom of the screen.

The *horizontal speed* is an integer from -128 to 127. A negative

horizontal speed will cause a sprite to move toward the left side of the screen. A positive *horizontal speed* will cause a sprite to move toward the right side of the screen.

The absolute value of the *vertical speed* and the *horizontal speed* determine the speed of the sprite. If the absolute value of the specified number is close to zero, the sprite will move slowly. If the absolute value of the specified number is much greater than zero, the sprite will move quickly.

Example

```
100 CALL CLEAR :: CALL MAGNIFY(2)
200 CALL CHAR(92,"7090284482FF7E3C")
300 CALL HCHAR(15,1,94,32)
400 CALL SPRITE(#1,92,5,101,30,0,10)
500 GOTO 500
```

The preceding example contains a program that uses a sprite. When the program is executed, a small sailboat appears to be moving across the screen.

Line 100 of the program will clear the screen and magnify all sprites. Line 200 will redefine character 92 into the shape of a sailboat. Line 300 will place a row of carets (^) across the screen. Line 400 will create sprite #1 using character 92. The color of the sprite will be blue. The sprite will begin at position 101,30 and proceed with a horizontal speed of 10. Line 500 will prevent the program from ending.

When the program is executed, a sprite with the shape of a sailboat will move across the screen.

To stop the program, press FCTN 4 (CLEAR).

CALL VCHAR

- EX BASIC
- ST BASIC

The CALL VCHAR statement accesses the subprogram VCHAR. The VCHAR subprogram is used to place characters at specified positions on the screen.

Configuration

CALL VCHAR(row, column, ASCII code [,repetition])

The first two parameters, *row* and *column*, determine the position where the character will be displayed. The screen is divided into 24 rows and 32 columns. Therefore, the specified row must be a number from 1 to 24 and the specified column must be a number from 1 to 32.

The third parameter, *ASCII code*, is the ASCII code of the character that is to be placed at the specified position.

The fourth parameter, repetitions, is optional. *repetition* must be a number from 1 to 32767. The *repetition* specifies the number of times that the specified character is to be repeated. The specified number of characters will be repeated vertically.

If no parameter is included for *repetition*, the character will be displayed only once.

Example

```
> CALL VCHAR(5,15,65,10)
```

The preceding example contains a CALL VCHAR statement that will display a column of ten letter A's starting at position 5,15 on the screen.

CALL VERSION

■ EX BASIC
□ ST BASIC

CALL VERSION accesses the subprogram VERSION. The subprogram VERSION returns a value that corresponds to the type of BASIC being used. Extended BASIC returns a value of 110.

Configuration

CALL VERSION(a)

The parameter of the CALL VERSION statement is a numeric variable.

Example

```
>CALL VERSION(A)

>PRINT A
 110
```

CHR$

■ EX BASIC
■ ST BASIC

The CHR$ function returns the character with the ASCII code specified by the argument. The ASCII codes correspond to the values between 0 and 255. If the argument is not an integer, it will be rounded off.

Configuration

X$ = CHR$(a)

Example

```
>PRINT CHR$(65)
 A
```

CLOSE

■ EX BASIC
■ ST BASIC

The CLOSE statement closes a file that has been opened for input, output, or both. Closing a file that has not been opened will result in a FILE ERROR message. Closing a file causes the computer to no longer associate a particular filenumber with its corresponding file.

Configuration

CLOSE #a (:DELETE)

The filenumber must be a number between 1 and 255 inclusive. To close a particular file, the filenumber used in the CLOSE statement must be the same as the filenumber used in the corresponding OPEN statement.

Example

```
CLOSE #2
```

The CLOSE statment has a DELETE option. When used in a CLOSE statement, the DELETE option erases the file that was closed. The DELETE option can only be used to erase a disk file.

Example

```
CLOSE #1 : DELETE
```

CONTINUE (CON)

■ EX BASIC
■ ST BASIC

The CONTINUE command causes the execution of a program to resume after a breakpoint.

Configuration

CONTINUE

A breakpoint is an interruption in the execution of a program due to a BREAK statement or the CLEAR command (FCTN 4).

When the CONTINUE command is executed, the program execution resumes at the exact point where it was stopped. The CONTINUE command cannot be used in instances where an error was encountered or where program lines were added or edited. If an attempt is made to use the CONTINUE command in such a circumstance, the following message will be displayed.

*CAN'T CONTINUE

Example

```
>100  A=7
>200  PRINT A
>300  BREAK
>400  PRINT A
>RUN
   7
   * BREAKPOINT IN 300
>CON
   7
```

Notice in the preceding example that even after the breakpoint, the computer retained the value of the variable A.

Variables will retain their values after a breakpoint. Variables will be cleared, however, if the program is edited in any way.

COS

■ EX BASIC
■ ST BASIC

The COS function returns the cosine of its argument. In order to obtain the correct answer, the argument must be in radians.

Configuration

X = COS(*a*)

Example

```
>100  Y = 3.14159269
>200  X = COS(Y)
>300  PRINT X
>RUN
  -1
```

DATA

- EX BASIC
- ST BASIC

The DATA statement supplies information that can be assigned to variables through READ statements. A DATA statement can include numeric values, string values or both.

DATA items are separated by commas. Therefore, string values that contain commas will be read as separate data items. For example, DATA DOE, JOHN is a DATA statement with 2 data items. However, DATA DOE. JOHN has only one item. Commas may be included in strings within a DATA statement if the string is enclosed in quotation marks ("). For example, the following DATA statement has only one data item. DATA "JOHN, STEVE, PAT, TOM".

Configuration

$$\text{DATA} \begin{matrix} a \\ a\$ \end{matrix} \left[\begin{matrix} , b \\ , b\$ \end{matrix} \right] \ldots$$

The DATA statement includes a list of constant values known as a DATA list. The items in the DATA list are assigned sequentially to the variables in the READ statement.

Therefore, the order in which DATA statements appear is very important.

The DATA list uses a pointer to indicate which value within the list is to be assigned to the next variable in a READ statement.

Before the first READ statement is ecnountered, the DATA list pointer will point at the first value in the DATA list. As values from the DATA list are assigned to variables in the READ statement, the pointer will move sequentially to each successive item in the DATA list.

Example

```
>100  FOR A = 1 TO 9
>200  READ B
>300  PRINT B;
>400  NEXT A
>500  DATA 9,8,7,6
>600  DATA 5,4,3,2,1
>RUN
   9 8 7 6 5 4 3 2 1
```

The values from the DATA list must match the type of variable to which they are assigned in the READ statement. For example, a string value cannot be assigned to a numeric variable. A numeric value can be assigned to a string variable, but this numeric value may no longer be used as a number but rather as a string.

Example

```
>100  FOR J = 1 TO 5
>200  READ A$
>300  PRINT A$
>400  NEXT J
>500  DATA TOM C. , 25,, 3 + 4 * %, 247
>RUN
  TOM C.
  25

  3 + 4 * %
  247
```

The preceding example shows correct data for a string variable. Notice the blank line in the output that corresponds to the two commas in a row. This is read as a string value with no characters and length equal to zero.

If only four data items had been supplied with this program, the message DATA ERROR IN 200 would have been displayed to notify the user that not enough data had been supplied.

Both standard and scientific notation are acceptable in a DATA statement. For example, 3.14159266, 2.85E-10, .0001, 35 and -45 are all acceptable DATA statement items that can be assigned to corresponding numeric variables in a READ statement.

Expressions such as 4 + 5, 12/6, and 10 * 5 will not be evaluated when assigned to a numeric variable. Instead, they will cause a DATA ERROR with the line number of the READ statement.

Example

```
100 FOR I = 1 TO 5
200 READ A$,A
300 PRINT A$,A
400 NEXT I
500 DATA PENCILS, 20, PENS, 25,
    RULERS, 40, ERASERS, 50, PAPER,
    200, GLUE, 5
```

The preceding example shows a correct sequence for reading string and numeric data into correct variables. However, the READ statement is only called 5 times, and there are 6 sets of data. This will not cause an error, but the last set of data (GLUE,5) will never be read.

Data can only be read once unless a RESTORE statement has been executed. A RESTORE statement causes the data pointer to return to a specified DATA statement.

DEF

- EX BASIC
- ST BASIC

The DEF statement allows a user to define a function that can be used repeatedly in a program.

Configuration

$$\text{DEF} \begin{matrix} \text{A} \\ \text{B\$} \end{matrix} \left(\begin{matrix} \text{X} \\ \text{Y\$} \end{matrix} \right) = \textit{function definition}$$

The DEF function can be used to manipulate string or numeric values. The type of values that are returned by the function depend on the nature of the function definition. However, the variable name that follows the keyword DEF must be the same type as the value returned by the function.

The variable name in a DEF statement that is enclosed in parentheses represents the function's argument. This variable name is used in the function definition to calculate the value returned by the function. The type of value used as an argument must agree with the type of variable used in the function definition.

Example

```
>10 DEF A$ (B) = CHR$(42)&CHR$(B)
>20 PRINT A$(65)
>RUN
 *A
```

The preceding example contains a program that defines a function. Line 10 contains a DEF statement that defines the function A$. Since the function name is a string variable, the function must return a string value.

In the DEF statement, the argument of the A$ function is the numeric variable B. As a result, the A$ function must always have a numeric value as an argument.

The A$ function returns an asterisk, followed by the character that corresponds to the ASCII code specified by the argument.

Illustration 5-5. Function Definition Examples

function definition	argument	result
DEF A$(B) = CHR$(B)	numeric	string
DEF A(B$) = ASC(B$)	string	numeric
DEF A(B) = SIN(B) + COS(B)	numeric	numeric
DEF A$(B$) = SEG$(B$,2,2)	string	string

Illustration 5-5 contains four simple DEF statements. The type of argument that each function requires are also listed. The last column describes the type of value returned by the function.

DELETE

- EX BASIC
- ST BASIC

The DELETE command is used to erase programs or files recorded on a storage device.

Configuration

DELETE "*device-name.prog-name*"

The DELETE command is generally used with a disk drive to delete programs and files. The DELETE command cannot be used with a cassette recorder.

Example

```
>DELETE "DSK1.PROG1"
```

The preceding example could be used to erase a program that had been previously stored on a disk. In this case, the device was a disk drive (DSK1) and the program name was PROG1.

DIM

■ EX BASIC
■ ST BASIC

In Standard BASIC, the DIM statement is used to set aside memory space for a 1, 2, or 3 dimensional string or numeric variable. In Extended BASIC, DIM can be used to set aside memory for an array of up to seven dimensions.

If an array variable is used in a program without having first been dimensioned, that variable will automatically be dimensioned to 11 elements.

Configuration (EX & ST BASIC)

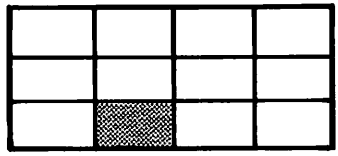
```
DIM X(a[,b] [,c])   [,Y(d[,e] [,f])]
    X$(a[,b] [,c])  [,Y$(d[,e] [,f])] ...
```

Configuration (EX BASIC)

```
DIM X(a[,b,c,d,e,f,g])   [,Y(b[,i,j,k,l,m,n])]
    X$(a[,b,c,d,e,f,g])  [,Y$(b[,i,j,k,l,m,n])]
```

The lowest element number of an array is automatically set to zero. This can be changed, however, using an OPTION BASE statement. The highest element number is specified in the DIM statement. For example, DIM X(100) would dimension a numeric array of 101 elements. The first element would be identified as X(0) and the last as X(100).

Variables can also be used with two or three subscripts in Standard BASIC, and up to seven subscripts in Extended BASIC. For example, if X is dimensioned by the statement DIM X(2,3), the following table will result.



The values in this table can be named by X with two subscripts (in parentheses). The first subscript is the row number, and the second is the column number. For example, the value of the shaded block would be X(2,1) because it is row 2, column 1.

Example

```
100 FOR H = 0 TO 1
110 FOR I = 0 TO 3
120 FOR J = 0 TO 2
130 READ X(H,I,J)
140 PRINT X(H,I,J);
150 NEXT J
160 PRINT
170 NEXT I
180 PRINT
190 PRINT
200 NEXT H
210 DATA 1,2,3,4,5,6
220 DATA 7,8,9,1,2,3
230 DATA 4,5,6,7,8,9
240 DATA 1,2,3,4,5,6
```

This example shows a technique for assigning values to tables. This particular example uses a three dimensional array. Note that since none of the subscripts exceed the number 10, a DIM statement is not necessary. Executing this program would have the following result.

```
1  2  3
4  5  6
7  8  9
1  2  3

4  5  6
7  8  9
1  2  3
4  5  6
```

A DIM statement can include any combiation of numeric and

string variable dimension statements. For example, DIM A(10,10), B(9), A$(90), B$(90) dimensions all four variables in one statement.

If a variable is going to be used with more than eleven subscripts, a DIM statement must appear in the program before the subscripted variable is used. For example, if X is being dimensioned to 20, DIM X(20) must appear in the program before the variable X is used.

A variable must always contain a subscript once it has been dimensioned. For example, if the statement DIM X(20) appeared in a program, all references to the variable X must be made in the form X(Y) where Y is an integer between 0 and 20.

DISPLAY

■ EX BASIC
■ ST BASIC

The DISPLAY statement is similar to the PRINT statement. However, a DISPLAY statement may not be used to write to any device other than the screen.

Configuration

DISPLAY (*expression*) (;) ...

Example

```
>200 READ A$, B
>300 DISPLAY A$; "IS"; B; "YEARS OLD"
>600 DATA TIM, 21
>RUN
 TIM IS 21 YEARS OLD
```

DISPLAY can be used in Extended BASIC in the same manner as it is used in Standard BASIC. However, in Extended BASIC, many more options are available.

Configuration

DISPLAY [AT(*row, column*)] [BEEP] [ERASE ALL]
[SIZE (*parameter*):] *expression*

The AT option is used to specify the position of the output on the screen. The data will be printed at the specified row and column. The screen is divided into 24 rows and 28 columns.

The BEEP option causes a short tone to be generated when the data is being printed.

The ERASE ALL option causes the entire screen to clear before the data is printed. Therefore, when the data is printed, nothing else appears on the screen.

Example

```
100 DISPLAY AT (6,6) BEEP ERASE ALL: "NAMES AND ADDRESSES"
200 FOR T = 7 TO 12
300 READ B$
400 DISPLAY AT (T,6):B$
500 NEXT T
600 DATA ,JOHN DOE, 4444 ROCKHAVEN
700 DATA ,JANE LANG, 6526 FRANKLEN
```

The first DISPLAY statement in the preceding example will clear the screen, generate a short tone, and print the phrase NAMES AND ADDRESSES starting at position 6,6 on the screen.

The second display statement prints the actual names and addresses.

When executed, the preceding example will generate the following output.

NAMES AND ADDRESSES

JOHN DOE
4444 ROCKHAVEN

JANE LANG
6526 FRANKLEN

Another option of DISPLAY is the SIZE option. SIZE determines the number of spaces on the line to be cleared for the data. If the SIZE option is not used, an entire line will be cleared.

Example

```
>10 A$ = "WILLIAM JONES"
>20 DISPLAY AT(23,5) SIZE(7):A$
>RUN WILLIAM <------ program output
```

The preceding example contains a program that uses the AT and SIZE options with a DISPLAY statement. At line 10, the variable A$ is assigned a value. At line 20, a DISPLAY statement specifies screen location (23,5). This location is 5 spaces from the left margin on the last line of the display. The SIZE option specifies that only seven spaces are available for the output.

When the RUN command is executed, the output is displayed one space to the right of the keyword RUN. Only the first seven characters of the string are actually output. The value of A$ is truncated because of the SIZE option of the DISPLAY statement.

DISPLAY USING

■ EX BASIC
□ ST BASIC

By including the keyword USING in a DISPLAY statement, a printing format may be specified. The printing format specifies the configuration of the output.

Configuration

DISPLAY [*options*:] USING $\left\{\begin{matrix}\text{"}\textit{expression}\text{"} \\ \textit{line number}\end{matrix}\right\}$ [:*data list*]

The *options* specified after DISPLAY are the same as all the options discussed previously (AT, BEEP, ERASE ALL, and SIZE). The *expression* after the USING statement consists of special characters that are used in defining the format in which the *data list* is to be displayed. These special characters are listed in Table 5-4 on page 194.

Illustration 5-5. Values, Format Strings and Results

VALUE	FORMAT	RESULT
7	$##.##	$ 7.00
667	###.#	667.0
5674	##	6E+03
994	TI###A	TI994A
235	$###	$235
8757	###	***
8.235	#.#	8.2
9	###	9
78	+##	+78
1.9	#	2
.999	#	1

Illustration 5-5 shows how different formats will effect the display of different numbers.

A *line number* can also be included in a USING statement. *Line number* must be the program line number of an IMAGE statement. The IMAGE statement is used to define a format that can be used with more than one DISPLAY or PRINT statement.

EDIT

□ EX BASIC
■ ST BASIC

The EDIT command displays a line for editing. The EDIT command also instructs the computer to enter the Edit Mode. In the Edit Mode, existing program lines can be changed or deleted.

Configuration

EDIT *line number*

An alternative method for entering the Edit Mode (other than executing EDIT) is to enter a line number and then press either FCTN ↑ or FCTN ↓.

The EDIT command cannot be used in Extended BASIC. The Edit Mode can only be entered by using the FCTN keys as previously explained. Once in the Edit Mode, program lines are edited in the same fashion, regardless of the version of BASIC.

In the Edit Mode, the specified program line will be displayed with the cursor at its beginning. By using special function keys, the cursor can be moved along the line to be edited, and changes can be made. The only portion of the line that cannot be changed is the line number.

The following keys are used in the Edit Mode.

ENTER — The ENTER key will enter the line, whether or not it has been changed, into the computer's memory. ENTER will also cause the computer to leave the Edit Mode.

FCTN ↑ — This key causes the line being displayed for editing to be entered into memory. The line with the next lowest line number is then displayed for editing. If a line with a lower line number does not exist, the computer will leave the Edit Mode.

FCTN ↓ — This key causes the line being displayed for editing to be entered into memory. The line with the next highest line number is then displayed for editing. If a line with a higher line number does not exist, the computer will leave the Edit Mode.

FCTN ← — This key is used to move the cursor to the left. The cursor will move backward in the line without changing or deleting any characters.

FCTN → — This key is used for moving the cursor to the right. The cursor will move forward along the line without changing or deleting any characters.

FCTN 2 (INSERT) — This key is used to insert characters in a program line. To use this key, move the cursor (using FCTN → and FCTN ←) to the point where additional characters are needed. Then, depress the INSERT key and proceed by typing the additional characters. Any characters to the right of the cursor will automatically move to the right as characters are inserted. Once the addition has been made, the line can be entered into memory using either ENTER, FCTN ↑, or FCTN ↓.

FCTN 1 (DELETE) — This key is used to delete any unwanted characters from a line. To delete a character or set of characters, move the cursor (using FCTN → and FCTN ←) to the leftmost character to be deleted. Once the cursor is over the character to be deleted, depress FCTN 1. The computer will delete the character and automatically move everything to the right of the cursor one space to the left.

FCTN 4 (CLEAR) — This key causes the computer to leave the Edit Mode. However, any changes made in the line being edited will not be stored. The line being edited when the CLEAR key is used will not be altered. It will be stored in its original form.

FCTN 3 (ERASE) — This key erases the entire line being edited. The only portion of the line not erased is the line number.

END

■ EX BASIC
■ ST BASIC

An END statement ends the execution of the program. An END is not necessary at the end of a program because execution stops automatically after the last statement. However, it is a good programming technique to end BASIC programs with an END statement.

Configuration

END

When an END statement is executed, all files opened in the program will be closed.

Example

```
100 INPUT X
200 IF X <= 10 THEN 500
300 PRINT "X IS LARGER THAN 10"
400 GOTO 100
500 END
```

The program in the previous example will end only if a value of X is entered which is less than or equal to 10.

EOF

■ EX BASIC
■ ST BASIC

The EOF (End Of File) function is used to determine if the end of a file has been reached.

Configuration

EOF(*filenumber*)

The *filenumber* in the EOF function must correspond to a filenumber used in an OPEN statement.

If the end of the file has been reached, the EOF function will return a value of 1. If the end of file has not been reached, the EOF function will return a value of 0. If the end of file has been reached due to mechanical reasons such as a full disk or mechanical failure, the EOF function will return a value of -1.

Example

```
100 OPEN #1: "DSK1.TEST", INPUT, FIXED
200 IF EOF(1) THEN 9999
```

The preceding example contains a portion of a program that uses the EOF function to transfer program control to line 9999 if the end of the file has been reached.

EXP

- EX BASIC
- ST BASIC

The EXP function returns the exponential of the argument. The exponential is the approximate value of e (2.718281828) raised to the power of the argument.

Configuration

X = EXP(a)

Example

```
>PRINT EXP(5)
 148.4131591
```

■ EX BASIC
■ ST BASIC

FOR

A FOR statement is used with a NEXT statement to form a repetitive loop within a program.

Configuration

FOR A = *a* TO *b* [STEP *c*]

Every FOR statement must have a corresponding NEXT statement.

Example

```
>10 FOR J = 1 TO 5
>20 PRINT J;
>30 NEXT J
>RUN
   1  2  3  4  5
```

In the previous example, the FOR/NEXT loop is repeated five times. Line 20 is the only statement inside the loop, however, any number of program lines can be placed within a loop.

In line 10, J is assigned the value 1. J is referred to as a counter. The value of J is incremented when a NEXT J statement is executed. Here, the program returns to the FOR statement, where J is incremented by one. This loop is repeated until J is set equal to 5. When the counter (J) has been set equal to the final value (5), and the loop has been executed, the program will proceed with the statement following NEXT J.

A FOR/NEXT loop can use a STEP statement to increment the counter by a value other than 1.

Example

```
>10 FOR J = 1 TO 2 STEP .5
>20 PRINT J;
>30 NEXT J
>RUN
   1  1.5  2
```

The preceding example contains a FOR/NEXT loop that increments the value of J by .5 each time the loop is executed.

A FOR/NEXT loop can also be used to decrease the value of the counter. This can be accomplished by using the optional STEP statement within the FOR statement. If the STEP statement has a negative argument, the counter is decreased each time the loop is executed. The following example illustrates a FOR/NEXT loop in which the counter is decremented rather than incremented.

Example

```
>10 FOR K = 10 TO 5 STEP -2
>20 PRINT K;
>30 NEXT K
>RUN
   10  8  6
```

This loop begins at line 10 by assigning the counter (K) the value 10. At line 20 the value of K is printed. When line 30 is encountered, execution continues at line 10, because the NEXT statement returns the program to the preceding FOR statement. The value of the counter is changed by the argument of STEP. Since the STEP value is -2, the counter is decreased by 2. The value of the counter is changed to 8. At line 20, the new value of K is printed. Line 30 is executed again, so the program returns to the FOR statement at line 10. The counter is again decremented by 2. The new value of K is 6. At line 20, this K value is printed.

When line 30 is executed again, the program does not return to line 10. The current value of the counter is 6, and if the counter was to be decremented again, the counter would be 4. However,

4 is less than the final value specified in the FOR statement (the argument of TO). As a result, the loop does not continue after K = 6 because another decrement would make the counter less than the final value (5).

If the counter of a loop is being incremented, the loop will be executed until the counter would exceed the final value if it were incremented again. For example: FOR J = 1 TO 4 STEP 2 would be executed with J equal to 1 and 3. The counter (J) would exceed the final value (4) if it were incremented again.

A FOR/NEXT loop should be executed as if it were a single statement. An attempt to branch into a FOR/NEXT loop will cause an error.

Example

```
>10 GOTO 30
>20 FOR T=1 TO 10
>30 PRINT T
>40 NEXT T
>RUN
  0

 * CAN'T DO THAT IN 40
```

In general, branching out of a FOR/NEXT loop will not cause an error. However, exiting a loop before it has completed should be avoided.

GOSUB

■ EX BASIC
■ ST BASIC

GOSUB branches program control to the subroutine beginning at the line number specified by its argument.

Configuration

GOSUB *ln*

Subroutines can be called from any part of a program. A RETURN statement, at the end of a subroutine, causes the program to resume execution with the statement directly after the GOSUB statement.

Subroutines are convenient to use when the same set of operations need to be repeated at different parts of a program.

Example

```
>10 FOR J = 0 TO 2
>20 GOSUB 100
>30 NEXT J
>40 J = 5
>50 GOSUB 100
>60 END
>100 PRINT J;
>110 RETURN
>RUN
  0  1  2  5
```

The previous example illustrates a subroutine that is called 4 times, from 2 different parts of the program. In this example, only one statement is included in the subroutine. However, many statements can be included in a subroutine.

Line 10 begins a FOR/NEXT loop. The counter (J) is set equal to 0 the first time through the loop. Line 20 calls the subroutine at line 100. As a result, line 100 is executed next. The subroutine prints the value of J and proceeds to line 110. At line 110, the program is returned to the point where the subroutine was called (line 20).

The statement at line 30 is then executed. The NEXT statement causes the loop to be incremented and repeated. The counter (J) is set equal to 1, and the subroutine is called again from line 20. At line 100, the value of J is printed. Line 110 returns the program to line 20.

These steps are also repeated for J = 2. When the loop has been executed 3 times, the program will proceed to line 40. J is assigned the value 5, and the subroutine is called again at line 50. The subroutine prints the value of J. The program then returns to line 60 where it ends.

GOSUB can also be used with ON to branch a program to one of several subroutines.

Configuration

ON EX GOSUB *ln* [*ln*, ...]

The expression after the ON statement indicates which line number the program proceeds to. This is called the control expression. The control is evaluated and rounded off. If the value is negative, an error occurs. If the value of the control is 1, the program continues at the first line number after GOSUB. If the control is equal to 2, the program continues at the second line number after GOSUB, etc.

If the value of the control is 0 or greater than the number of line numbers, an error occurs.

Example

ON X GOSUB 100, 200, 300, 400

This statement executes the subroutine at line 100 if X = 1. If X = 2, the subroutine at line 200 is executed. If X = 3, the subroutine at line 300 is executed. If X = 4, the subroutine at line 400 is executed. If X = 0 or X is greater than 4, an error occurs.

GOTO

- EX BASIC
- ST BASIC

The GOTO statement causes the program to procceed at the indicated line number.

Configuration

GOTO *ln*

Example

```
>10 X = X+1
>20 IF X^2 > 50 THEN 50
>30 PRINT X;
>40 GOTO 10
>50 END
>RUN
  1  2  3  4  5  6  7
```

The previous example demonstrates the use of GOTO. Line 10 increases the value of X by 1. Line 20 ends the program when X squared is greater than 50. When line 40 is executed, the program returns to line 10. This program repeats lines 10 through 40 until the program is ended or branched out of the loop. The program ends when X = 8 because 8 squared is greater than 50.

GOTO is also used with an ON statement to branch a program to one of several lines.

Configuration

ON *ex* GOTO ln [,*ln*, ...]

The expression after the ON statement indicates which line number the program proceeds to. This is called the control expression. The control is evaluated and rounded off. If the value is negative, an error occurs. If the value of the control is 1, the program continues at the first line number after GOTO. If the value is 2, the program continues at the second line number after GOTO, etc.

Example

```
10 FOR J = 1 TO 3
20 ON J GOTO 30,50,70
30 PRINT "J = 1"
40 GOTO 80
50 PRINT "J = 2"
60 GOTO 80
70 PRINT "J = 3"
80 NEXT J
```

The previous example is a program that uses an ON/GOTO branch. Line 10 begins a FOR/NEXT loop that is repeated 3 times. The first time through the loop, the counter (J) is set equal to 1. At line 20, the program is branched to the first line number after GOTO because the control (J) is equal to 1. At line 30, the message J = 1 is printed. The program is sent to line 80, where the NEXT J is chosen. Since the loop increments the counter by 1, the counter is set equal to 2 during the second execution of the loop. At line 20, the program is branched to line 50, the second line number after GOTO. At line 50, the message J = 2 is printed. The loop is repeated again with J set equal to 3. At line 20, the program branches to line 70, the third choice. At line 70, the message J = 3 is printed. The loop is complete so the program ends.

If the control expression (argument of ON) equals zero or a number greater than the number of choices, an error occurs.

IF

■ EX BASIC
■ ST BASIC

The IF statement is used with a THEN statement to branch a program if a particular condition is true.

Configuration (EX & ST BASIC)

IF *expression* THEN *ln* [ELSE *ln*]

The *expression* that follows IF can be logical, relational or algebraic. Any of the logical, relational or algebraic operators can be used in this expression.

The expression in an IF THEN statement is used to control the program flow. If the expression is true, the program branches to the line number that follows THEN. If the expression is false, the program branches to the line number that follows ELSE.

If an ELSE statement is not included with IF and THEN, a false expression causes the program to proceed to the next line.

Example

```
100 IF X > Y THEN 600 ELSE 1000
```

The preceding example statement uses the expression X > Y to control the branching of the program. If the value of the variable X is greater than the value of Y, program control branches to line 600. If the value of X is less than or equal to the value of Y, program control branches to line 1000.

IF THEN ELSE statements can be used in Extended BASIC in the same manner in which they are used in Standard BASIC. However, Extended BASIC allows additional statements to be included within the IF THEN ELSE statement.

Configuration (EX BASIC)

IF *expression* THEN { *statement* [::*statement*] ... / *ln* } ELSE [*statement* [::*statement*] ... / *ln*]

Extended BASIC allows the use of both line numbers and statements after THEN and ELSE. Multiple statements can also be included. However, some statements may not be used with IF THEN ELSE. These statements are DATA, DEF, DIM, FOR, NEXT, OPTION BASE, SUB and SUBEND.

Example

```
100 IF X > Y THEN PRINT X::GOTO 500 ELSE PRINT Y::END
```

The previous example shows that multiple statements can follow a THEN or ELSE statement in Extended BASIC. When more than one statement is used, be sure to separate the individual statements with a double colon (::).

If the condition in the IF THEN ELSE statement is true, the statements following the THEN statement will be executed. If the condition is false, the statements following the ELSE statement will be executed.

■ EX BASIC
□ ST BASIC

IMAGE

The IMAGE statement specifies the format used when displaying data. The IMAGE statement does not cause any data to be output, but it defines a format that can be used to output data with a PRINT USING or DISPLAY USING statement.

Configuration

IMAGE "*expression*"

The *expression* within all IMAGE statements is of the same format as the expression used in the PRINT USING and DISPLAY USING statements. The expression consists of special characters used to define the format of the output. These special characters are listed in Table 5-4.

In general, any string can be placed within the quotation marks. The string will appear in the output exactly as it does in the IMAGE expression.

Table 5-4. Special Format Characters

#	specifies a digit position. If the number being printed has fewer digits than allowed by the formatting characters, the number will be right justified in the formatting field. If the number being printed has more digits to the left of the decimal point than allowed by the formatting characters, several asterisks (*) will appear in place of the number.
	a decimal point may be inserted anywhere in the format field. It will appear in the number as it does in the format field.
^^^^	when four or five carats are placed at the end of the format string, scientific notation is used. The largest number which can be displayed using ^^^^^ is 9.9999999E + 127.

The following example demonstrates the use of an IMAGE statement.

Example

```
>10 IMAGE "COST: $##.##"
>20 PRINT USING 10: 23.475
>RUN
 COST: $23.48
```

Note that the value that is output is not the same value that appears in the PRINT USING statement. The value is rounded off in order to achieve the format defined in line 10.

Although Extended BASIC offers multiple statement lines, the IMAGE statement must be the only statement on a particular line. If an IMAGE statement is not alone on a line, an error generally will not occur, but the program may not work properly.

INPUT

- EX BASIC
- ST BASIC

The INPUT statement permits data entry from the keyboard while the program is being executed.

Configuration

$$\text{INPUT}\ [\text{"}prompt\text{"}:]\left\{\begin{array}{c} X[,Y]\ldots \\ X\$[,Y\$]\ldots \end{array}\right\}$$

When an INPUT statement is encountered, a question mark appears on the display as a signal for the operator to enter data. A prompt message can also be included to print a message when an INPUT statement is executed. However, if a prompt is included, the question mark does not appear on the display.

An INPUT statement can include a combination of numeric and string variables. Values can be assigned to each variable with one response. The values of a multiple response must be separated by commas. The response to an INPUT statement must always be ended by pressing the ENTER key.

The following warning message is dispalyed if the number of values entered does not match the number of variables in the corresponding INPUT statement.

```
*WARNING
 INPUT ERROR IN line number
```

This previous warning is also generated if a string value is entered where a numeric value is expected.

Example

```
>100 INPUT "ITEM, QTY.? ":ITEM$, QTY
>200 PRINT ITEM$;QTY
>RUN
 ITEM, QTY? PENS, 1200      ←user's response
 PENS 1200
```

The preceding example demonstrates the use of an INPUT statement to assign values to two variables. When the program is executed, the prompt "ITEM, QTY?" is displayed on the screen. The computer then waits for an appropriate response from the keyboard.

The two values that are entered are PENS and 1200. The data that is entered is displayed on the screen immediately to the right of the prompt. The PRINT statement at line 200 causes the values of the two variables to be displayed on the screen.

If a string value that contains one or more commas is used in response to an INPUT statement, the string must be enclosed in quotation marks. Otherwise, the commas will be interpreted as delimiters instead of characters in the string. When string values

are enclosed in quotation marks, the quotation marks are not considered part of the string.

INPUT

- EX BASIC
- ST BASIC

The INPUT # statement is used to read data from an input device, and assign the values to variables.

Configuration

INPUT #*filenumber* [,REC *record number*]: *variable* [,*variable*] ...

The *filenumber* is the number assigned to the appropriate input file. The *filenumber* must have been used in an OPEN statement prior to being used in an INPUT # statement.

The specified *variable* is assigned the value of the data item input from the appropriate device.

The *record number* is used with relative files to specify the position of the data pointer. The data pointer determines the position of the current data item in a relative file.

The data items being read and assigned to variables may be from a sequential file on cassette, from a sequential or relative file on diskette, from a communications device, or from the keyboard.

The type of data being read from the file must agree with the specified variable type.

When numeric items are being read, any leading blank spaces will be ignored. Numeric data can only include the digits 0 through 9, and the following special characters when appropriate.

. + – E

Leading blank spaces are ignored when string data is being input.

Quotation marks may not be used within a sring, but may be used to enclose a string if necessary. Generally quotation marks are needed only if a string contains a comma.

Example

```
100 INPUT # 1, REC 3 : A$
```

The previous example contains an INPUT statement that will retrieve data from file number one starting at record number 3. This data will be assigned to the variable A$.

INT

■ EX BASIC
■ ST BASIC

The INT function returns the largest integer that is less than or equal to the argument.

Configuration

X = INT (*a*)

Examples

```
>PRINT INT (13.9)
 13
>PRINT INT (-4.7)
 -5
```

LEN

■ EX BASIC
■ ST BASIC

The LEN function returns the number of characters in a string value or variable, including spaces and punctuation.

Configuration

X = LEN (*string*)

Example

```
>100 A$ = "JONES, BILL"
>200 PRINT LEN (A$)
>300 PRINT LEN ("BILL JONES")
>RUN
   10
   10
```

Line 100 assigns A$ a string value. Line 200 displays the number of characters in the variable A$. Line 300 displays the number of characters in the string "BILL JONES".

LET

■ EX BASIC
■ ST BASIC

The LET statement is optional. It is used to assign a value to a variable.

Configuration (EX & ST BASIC)

$$[\text{LET}] \begin{Bmatrix} A \\ A\$ \end{Bmatrix} = \begin{Bmatrix} x \\ x\$ \end{Bmatrix}$$

The type of variable in an assignment statement must correspond to the value of the expression. For example, if the variable is a string, the value of the expression must also be a string. If the variable is numeric then the value of the expression must be an integer or real number.

Any type of expression that returns an appropriate value can be used in an assignment statement.

Examples

```
LET X = 250
X = Y + 25
A = B > C
A$ = "TIM M."
```

Extended BASIC allows one value to be assigned to more than one variable simultaneously.

Configuration (EX BASIC)

$$[\text{LET}] \left\{ \begin{matrix} X[,Y...] \\ X\$[,Y\$...] \end{matrix} \right\} = \left\{ \begin{matrix} a \\ a\$ \end{matrix} \right\}$$

Examples

```
LET A,B,C,D,E = 10
A$,B$,C$,D$ = "TIM"
```

LINPUT

■ EX BASIC
□ ST BASIC

The LINPUT statement is used to assign an entire line of data to a string variable.

Configuration

LINPUT ["*prompt*":] *string variable*

The LINPUT statement will assign an entire line of text to the specified *string variable*. The data assigned to the string variable will include spaces, quotation marks, and commas. A line of input will be ended when the Enter key is pressed.

A prompt message can be included to print a message when an LINPUT statement is executed.

Example

```
>100 LINPUT A$
>200 PRINT A$
>RUN
  ? OUTPUT 23-35,40      —user's response
 OUTPUT 23-35,40
```

■ EX BASIC
□ ST BASIC

LINPUT

The LINPUT # statement is used to assign an entire line of data from an input device to a string variable.

Configuration

LINPUT # *filenumber* [,REC *record number*]: *string variable*

The *filenumber* is the number assigned to the appropriate input file. The *filenumber* must have been used in an OPEN statement prior to being used in a LINPUT # statement.

The specified *string variable* will be assigned the data item that was input from the file.

The *record number* is used with relative files to specify the location of the data pointer. The data pointer determines the position of the current data item in a relative file.

Example

```
100 LINPUT # 1, REC 3:A$
```

The previous example contains a LINPUT # statement that will assign data from file number 1 to the string variable A$. The data in record number 3 of the relative file will be assigned to the variable specified in the LINPUT # statement.

The LINPUT # statement must be preceeded by an appropriate OPEN statement for the specified *filenumber.*

LIST

■ EX BASIC
■ ST BASIC

The LIST statement is used to display the contents of the computer's memory.

Configuration

LIST [*device*:] [*ln*] — [*ln*]

The arguments of a LIST statement are optional. If the LIST statement is executed without any arguments, the entire program that is currently in memory will be displayed on the screen.

The arguments of a LIST statement can be used to specify a particular section of the program. If only one number appears in a LIST statement, the specified line of the program will be displayed. If there is no corresponding line number in the program, the line with the next highest line number will be displayed. If a line number is specified that is greater than any line in the program, the line with the largest line number will be displayed.

If a LIST statement has one argument, followed by a dash, every line of the program with a line number greater than or equal to the argument will be displayed.

If a LIST statement has one argument preceded by a dash, every line of the program with a line number less than or equal to the argument will be displayed.

Examples

```
LIST 100
LIST 10-100
LIST 10-
LIST -100
```

The first statement in the preceding example displays line 100. The second example displays all the lines in the program numbered from 10 to 100 inclusive. The third example displays line number 10 along with all the subsequent lines of the program. The fourth example displays all the lines of the program in memory up to and including line number 100.

A listing can be stopped in both Standard BASIC and Extended BASIC by pressing FCTN 4 (CLEAR). However, Extended BASIC allows a listing to be stopped and restarted again. To interrupt the listing of an Extended BASIC program, hold down any key until the listing stops. To restart the listing, simply press any key on the keyboard.

By including a proper device name after the list command, programs can be written to devices other than the screen. Proper device names include "TP", "DSK1" and "RS232".

Examples

```
LIST "RS232" : 100 — 200
LIST "TP"
```

The first statement in the preceding example sends a listing of lines 100 through 200 to the RS232 interface. The second statement sends a listing of the entire program in memory to the thermal printer.

LOG

■ EX BASIC
■ ST BASIC

The LOG function returns the natural logarithm of the argument. The natural log function is undefined for arguments less than or equal to zero.

Configuration

X = LOG (*a*)

Examples

```
>PRINT LOG (2.718281829)
  1.
>PRINT LOG (-1)
 * BAD ARGUMENT
```

A BAD ARGUMENT message is displayed when a zero or negative argument is used.

MAX

■ EX BASIC
□ ST BASIC

The MAX function returns the largest of its two arguments.

Configuration

X = MAX (*a*, *b*)

Example

```
>PRINT MAX (8, 3)
  8
>A = 15 :: B = 27 :: PRINT MAX (A, B)
  27
```

MERGE

■ EX BASIC
□ ST BASIC

MERGE is used to recover programs that have been saved on disk. MERGE can only be used to load programs that were recorded on disk with the SAVE command including the MERGE option.

Configuration

MERGE *device name . file name*

When the MERGE statement is executed, the computer's memory will not be erased. The new program being loaded will be placed in memory together with any existing program lines. For example, if the program in memory contained line numbers 10, 20, 30..., and the program being loaded (using MERGE) contained line numbers 5, 15, 25, 35,..., the resulting program in RAM would include the line numbers from each of the two programs.

MERGE does not alter the program in memory unless the program being merged has the same line numbers as the program in memory. For example, if the program in memory contains line numbers 10, 20, 30, 40, 50, and 60 and the program being merged contains 10, 20, 30, 45, 55, 70, 80, and 100, the new program in memory will contain all of the newly entered program, but only lines 40, 50, and 60 of the original program. The original lines 10, 20, and 30 will be replaced with lines 10, 20, and 30 being loaded from disk. Lines 40, 50, and 60 of the original program remain unchanged.

MERGE is the only statement that can recover a program without clearing the memory first.

Example

MERGE DSK1.TEST

The preceding example demonstrates the format of a MERGE statement. The disk file TEST is merged with the program that currently resides in the computer's memory. Before a program can be merged, it must first be saved with the MERGE option. The following SAVE statement demonstrates the format used to save a program that is intended to be merged.

```
>SAVE DSK1.TEST,MERGE
```

A program saved in this manner cannot be recovered with the OLD command. Only MERGE can be used to reload programs that are saved with the MERGE option.

MIN

■ EX BASIC
□ ST BASIC

The MIN function returns the smallest of its two arguments.

Configuration

X = MIN (*a*, *b*)

Examples

```
>PRINT MIN (16, 7)
 7
>A = 6 :: B = 3 :: PRINT MIN (A, B)
 3
```

NEW

■ EX BASIC
■ ST BASIC

The NEW command eliminates the current program in the computer's memory. The NEW command also erases all variables and closes any open files.

Configuration

NEW

Example

```
>NEW
```

NEXT

■ EX BASIC
■ ST BASIC

The NEXT statement is used with a FOR statement to form a repetitive section of a program.

Configuration

NEXT X

A FOR statement begins a loop, and a NEXT statement ends it. The FOR statement sets an initial value and a final value for the counter. The optional STEP statement specifies the amount that the counter is increased or decreased each time the loop is executed.

Example

```
10 FOR J = TO 10 STEP 2
20 PRINT J
30 NEXT J
```

In the previous example, the variable J is the counter. The initial value of the counter is 1, and the final value is 10. The value of the counter is incremented by 2 each time the loop is executed.

The section of the program between the FOR and NEXT statements is repeated for each different value of the counter. Each time the NEXT statement is executed, the value of the counter is changed by the STEP argument value. The loop is repeated for each value of the counter. In the previous example, the loop is repeated 5 times, with the counter equal to 1, 3, 5, 7, and 9. The initial value of the counter (J) is 1, and it is increased by 2 each time the loop is executed because of the STEP 2 statement.

If no STEP statement is used, the counter value increases by 1 each time a NEXT statement is executed.

A FOR/NEXT loop can also have a decreasing counter. If the STEP argument is negative, the value of the counter decreases each time the loop is executed.

An increasing counter will repeat the loop until one more increase would make the counter greater than the final value. A decreasing counter will repeat the loop until one more decrease would make the counter less than the final value.

When a loop has been completed, the statement after the NEXT statement will be executed.

NOT

■ EX BASIC
□ ST BASIC

The NOT statement is used as logical negation.

Configuration

NOT *ex*

The NOT operation is executed according to the following truth table.

A	NOT A
0	-1
-1	0

The conditions of true and false are represented by the values -1 and 0.

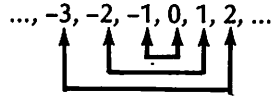
NOT statements are generally used in IF/THEN statements.

Example

```
10 Y = 5
20 IF NOT Y = 4 THEN 999
30 PRINT "Y IS EQUAL TO 4"
999 END
```

The program in the preceding example contains a NOT statement as part of an IF/THEN statement. At line 10, the variable Y is assigned the value 5. At line 20, the statement NOT Y = 4 is the condition of the IF/THEN statement. Since the value of Y is 5, the statement 5 = 4 is false. As a result, NOT Y = 4 is true, and program control branches to line 999. When Y has any value other than 4, the program has no output.

A NOT statement can also have a numeric argument. The NOT operation returns a numeric value according to the formula NOT A = (-A) -1. A numeric argument for a NOT statement is always rounded off so a NOT statement is always evaluated to an integer. The NOT operation of integers can be represented by the following illustration.



NUMBER (NUM)

■ EX BASIC
■ ST BASIC

The NUMBER command results in a new line number being generated every time the user types in a program line and presses the ENTER key.

Configuration

NUMBER [*number*] [,*increment*]

The specified *number* determines the first line which will be generated. The specified *increment* determines the amount to be added to the previous line number to generate the next line number.

The NUMBER command is generally used when entering programs. This saves the user the task of typing the line numbers.

If no parameters are included in the NUMBER command, the beginning line number will be 100 and each new line number will be incremented by 10.

If a beginning line number is specified in the NUMBER command, but the increment is not specified, the increment will be 10. If an increment is specified without an original line number, the first line number will be 100. Be sure to precede the increment value by a comma when a beginning line number is not specified.

When the NUMBER is in effect, a line number that already exists in the program may be generated. When the number of an existing program line is generated, the entire program line appears on the display. This line can then be changed, deleted or saved without any revisions. If the ENTER key is pressed before any changes are made, the line will remain in memory and a new line number will be generated.

There are two ways of exiting the NUMBER mode. If the user presses the ENTER key without first typing in a line, the computer returns to normal operation. The CLEAR command (FCTN 4) can also be used to exit the NUMBER mode. However, the current line being entered is erased when the CLEAR command is used.

Examples

```
NUM
NUM 100, 20
NUM 1000
NUM, 100
```

In the preceding example, the first command will generate the line numbers 100, 110, 120, 130 ... . The second command will generate the line numbers 100, 120, 140 ... . The third command will generate the line numbers 1000, 1010, 1020, 1030 ... . The line numbers generated by the final NUMBER command will be 100, 200, 300, etc.

OLD

- EX BASIC
- ST BASIC

The OLD command is used to retrieve programs that have been stored on disk or cassette.

Configuration

OLD *device name* [.*program name*]

Both a *device name* and a *program name* must be specified when retrieving programs from a disk. The proper *device names* for a disk drive are DSK1, DSK2,... The device name must correspond to the disk drive that contains the specified disk file. The *program name* that was used to save the program must also be used to retrieve the program.

Entering the OLD command causes the program with the specified *program name* to be loaded from the disk drive.

Example

OLD DSK1.PROGRAM1

The preceding example demonstrates the format of an OLD command. The *device name* (DSK1) specifies disk drive number 1, and PROGRAM1 is the name of the program.

Using the OLD command with a cassette recorder involves several steps. The command OLD CS1 must be entered, but a program name need not be specified.

When an OLD CS1 statement is executed, a series of prompts are displayed on the screen. These prompts guide the user through the steps required to operate the cassette recorder.

Example

```
>OLD CS1
 * REWIND CASSETTE TAPE      CS1
  THEN PRESS ENTER

 *PRESS CASSETTE PLAY      CS1
  THEN PRESS ENTER

 * READING

 * DATA OK

 * PRESS CASSETTE STOP      CS1
  THEN PRESS ENTER
```

The preceding example shows the prompts that appear when the OLD command is entered.

If an error occurs while the program is being loaded, an error message will appear.

ON BREAK

■ EX BASIC
□ ST BASIC

An ON BREAK statement determines the action that will be taken when a breakpoint is encountered in a program.

Configuration

ON BREAK { NEXT / STOP }

A breakpoint is an interruption in the execution of a program due to a BREAK statement or the CLEAR command (FCTN 4).

The keywords STOP and NEXT determine the action that will be taken when a breakpoint is encountered.

Using the keyword NEXT with the ON BREAK statement causes breakpoints to be ignored. When the ON BREAK NEXT statement is included in a program, commands such as BREAK 100 and FCTN 4 (CLEAR) are ignored. However, if a BREAK statement without an argument is encountered in a program, a breakpoint will occur regardless of the ON BREAK NEXT statement.

Using the keyword STOP with the ON BREAK statement causes program execution to stop when a breakpoint is encountered. During normal operation, the computer will always stop execution when a breakpoint is encountered. Therefore, the ON BREAK STOP statement corresponds to the default action of the computer.

An ON BREAK STOP statement is generally used to counteract the effect of an ON BREAK NEXT statement. Any number of ON BREAK statements may be included in a program.

Example

```
100 ON BREAK NEXT
program lines
200 BREAK 240, 260
program lines
300 BREAK
program lines
```

The program in the preceding example contains an ON BREAK NEXT statement and two BREAK statements. Because of the ON BREAK NEXT statement, the BREAK statement in line 200 will have no effect on the program. However, the BREAK statement in line 300 will cause the program to stop execution. The ON BREAK NEXT statement has no effect on line 300 since the BREAK statement is not followed by an argument.

When an ON BREAK NEXT statement is in effect, the program being executed will not be affected by the Function 4 (CLEAR) command. Normally the CLEAR command causes the execution of the program to be stopped.

ON ERROR

■ EX BASIC
□ ST BASIC

An ON ERROR statement determines the action that will be taken when an error occurs within a program.

Configuration

ON ERROR { *ln* | STOP }

The *ln* and the keyword STOP determine the action that will be taken when an error occurs.

When a line number is used as the argument of an ON ERROR statement, program control will be transferred to the specified line number when an error occurs.

An ON ERROR statement causes the transfer of program control to a subroutine in the same manner as a GOSUB statement. The execution of the subroutine begins at the specified line number and continues until a RETURN statement is executed.

The RETURN statement can be used to transfer program control back to the line that caused the error. The RETURN statement can also be used to transfer program control to any other line of the program.

Using the keyword STOP with the ON ERROR statement causes program execution to stop when an error occurs.

During normal operation, the computer will always stop execution when an error occurs. Therefore the ON ERROR STOP statement corresponds to the default action of the computer.

An ON ERROR STOP statement is generally used to counteract a previously executed ON ERROR statement.

Example

```
100 ON ERROR 500
200 INPUT A
300 PRINT SQR(A)
400 END
500 PRINT "TRY AGAIN"
600 RETURN 200
```

The preceding example contains a program that uses an ON ERROR statement. The statement at line 100 causes program control to branch to line 500 when an error occurs.

Lines 200 and 300 contain statements that accept numeric input from the keyboard, assign the value to the variable A, and print the square root of the value. The only type of data that can be assigned to the variable A are numeric values. As a result, an error occurs if the data input is not a numeric value.

At line 300, the SQR function is used to calculate the square root of the value of the variable A. If the value of the variable A is negative, an error occurs.

Lines 500 and 600 contain a simple subroutine that prints a message and returns program control to the INPUT statement at line 200.

ON WARNING

■ EX BASIC
□ ST BASIC

An ON WARNING statement determines the action that will be taken when a warning condition develops.

Configuration

```
                  ( PRINT )
ON WARNING        ( STOP  )
                  ( NEXT  )
```

The keywords NEXT, STOP, and PRINT specify the action that will be taken if a warning condition develops.

Warning conditions are ignored when an ON WARNING statement includes the keyword NEXT.

An ON WARNING PRINT statement causes the standard warning message to be displayed when a warning condition develops. Although a warning message is displayed, program execution will continue. The ON WARNING PRINT statement corresponds to the default action of the computer.

Using the keyword STOP with the ON WARNING statement causes program execution to stop when a warning condition develops. In the event of a warning condition, the standard warning message is displayed and program execution stops.

Example

```
> 100 ON WARNING NEXT
> 110 PRINT 10^1000
> 120 ON WARNING PRINT
> 130 PRINT 10^1000
> 140 ON WARNING STOP
> 150 PRINT 10^1000
> RUN
   9.99999E+**

 * WARNING
     NUMERIC OVERFLOW IN 130
     9.999999E+**

   * WARNING
     NUMERIC OVERFLOW IN 150
```

The previous example contains a program that handles the same warning condition in three different ways. The warning condition in each case is caused by a numeric overflow.

The first warning condition is ignored. The value displayed (9.99999E+**) indicates that overflow has occurred, but no warning message appears on the screen.

The second warning condition causes a warning message to be displayed. Although the warning message is displayed, program execution continues. The third warning condition causes a warning message to be displayed. In addition to the warning message being displayed, program execution will be stopped.

■ EX BASIC
■ ST BASIC

OPEN

An OPEN statement is used to open an I/O channel for an input or output device.

Configuration

OPEN #*filenumber*: "*device filename*"[,*format*][,*storage*][,*mode*][,*record length*]

The *filenumber* must be an integer from 1 through 255. This number is used as a reference for the particular file being opened.

The *device* is the I/O device that is being opened by the OPEN statement. Proper device names include "RS232", "DSK1", and "CS1".

The *filename* is a specific name used to reference a particular file.

Filenames need not be used with all devices. An RS232 and cassette recorder are two devices that do not require *filenames* when being opened. Disk drives, however, do require a *filename* when being opened.

The *format* specifies how files should be organized on the I/O device. The choices for *format* are RELATIVE and SEQUENTIAL.

Relative files allow any record to be accessed within that file without first searching all preceding records. Records within a sequential file can only be accessed in the order in which they appear in the file. The default value for *format* is SEQUENTIAL.

The *storage* specifies how the data will be stored. Data can be stored in two ways, binary form or ASCII form. Data stored in binary form can be accessed more quickly by the computer.

The keyword INTERNAL is used to specify binary storage. The keyword DISPLAY is used to specify ASCII storage. INTERNAL is generally the better choice when storing data on a disk or cassette. DISPLAY is generally the better choice when the output is to be sent to a printer. The default option for *storage* is DISPLAY.

The *mode* specifies the operation that will be performed while the file is open. The choices for *mode* are INPUT, OUTPUT, UPDATE, and APPEND. If INPUT is chosen, data can only be read from the file. If OUTPUT is chosen, data can only be written to the file. If UPDATE is chosen, the user may both read and write to a file. If APPEND is chosen, data may be added to a file without changing anything already in the file. The default option for *mode* is UPDATE.

The *record length* determines whether all the records will be the same length or will be of varying lengths. The options for *record length* are VARIABLE and FIXED. VARIABLE specifies that the records may be of varying lengths. FIXED specifies that all the records must be of the same length. The default option for *record length* depends on what other options were chosen. For example, the default *record length* for a RELATIVE file is FIXED, while the default *record length* for a SEQUENTIAL file is VARIABLE.

A maximum record length can be specified with the *record length* parameter. If a maximum record length is not specified the default values are 80 for diskettes, 64 for cassettes, 80 for the RS232 and 32 for the thermal printer.

Examples

```
OPEN #1:"DSK1.TEST",SEQUENTIAL DISPLAY, UPDATE, VARIABLE 80
OPEN #1:"DSK1.TEST"
```

The two example statements have the same effect. The first statement lists all the default values which are assumed in the second statement.

Either of the preceding examples will open file number one on disk drive 1. The name of the file is TEST. The file will be organized sequentially and have a display type storage. The user may read or write to the file, and records are of varying lengths with a maximum length of 80.

OPTION BASE

■ EX BASIC
■ ST BASIC

The OPTION BASE statement is used to specify a minimum value for array subscripts.

Configuration

OPTION BASE {1 / 0}

If an OPTION BASE of one is chosen, the lowest subscript that any array may have is one.

Only one OPTION BASE statement may appear in a program. The default value for OPTION BASE is zero.

Example

```
100 OPTION BASE 1
```

The maximum value of an array subscript can be specified with a DIM statement.

OR

■ EX BASIC
□ ST BASIC

The OR operator is generally used in IF/THEN statements to combine two or more conditions.

Configuration

ex OR *ex*

The OR operation is executed according to the following truth table.

A	B	A OR B
0	-1	-1
-1	-1	-1
-1	0	-1
0	0	0

The conditions of true and false are represented by the values -1 and 0.

Example

```
> 10 X = 5
> 20 Y = 10
> 30 IF Y > 5 OR X < 0 THEN 50
> 40 END
> 50 PRINT "Y IS GREATER THAN 5"
> 60 PRINT "OR X IS NEGATIVE"
> 70 END
> RUN
  Y IS GREATER THAN 5
  OR X IS NEGATIVE
```

In the preceding example, the variables X and Y are assigned values at lines 10 and 20. At line 30, an IF/THEN statement inlcudes an OR operator. The two expressions are Y>5 and X<0. Since Y>5 is true, and X<0 is false, the OR operation is evaluated as true.

Since the condition of the IF/THEN statement is true, the program control branches to line 50. At lines 50 and 60 the output message is printed. When the condition of the IF/THEN statement is false, the program is ended at line 40.

When the OR operation is executed with numeric expressions, the arguments are rounded off and converted to their binary equivalents. The value of the OR statement is the result of the OR operation on each bit of the values. For example:

```
      5 (binary 0101)
OR   11 (binary 1011)
-----------------------
     15 (binary 1111)
```

Since the binary equivalent of 5 is 0101, and the binary equivalent of 11 is 1011, the OR operation for each bit has the result 1111. The numeric value of 1111 is 15. Any non-zero value is considered true.

PI

■ EX BASIC
□ ST BASIC

The PI function returns the value 3.1415926539 (π).

Configuration

a = PI

PI is a physical constant that is used extensively in mathematic and scientific calculations.

Example

```
> PRINT COS (PI)
  -1.
> PRINT SIN (PI)
  0
```

POS

■ EX BASIC
■ ST BASIC

The POS function returns a value corresponding to the position of one string within another.

Configuration

X = POS (A$, B$, a)

The value returned by the POS function is the position of the first occurance of B$ in A$ starting at position a. If B$ does not occur in A$, a value of 0 is returned.

Examples

```
> PRINT POS ("STEPHEN", "E", 1)
  3
> PRINT POS ("STEPHEN", "E", 4)
  6
```

The preceding two examples show the proper use of the POS function. In the first example, the computer searches for the letter E in STEPHEN starting at the S (or position 1). In the second example, the computer searches for the letter E in STEPHEN starting at the letter P (or position 4).

PRINT

■ EX BASIC
■ ST BASIC

The PRINT statement is used to output characters to the display or an ouptut device.

Configuration

PRINT [[#a][,REC *record number*]:][*expression*](;)]...

The first argument of a PRINT statement is the optional file number. The file number is generally used only when an output device other than the display is used.

A PRINT statement can include string and numeric values. Each variable name or constant must be separated by either a comma, semicolon or colon. When a comma separates the items in a PRINT statement, the display is divided into two columns. When a semicolon is used between items in a PRINT statement, the values are displayed adjacent to each other, with one space preceding and trailing numeric values. When a colon is used between items in a PRINT statement, each item is printed on a separate line. An entire line is skipped for each colon that appears between data items.*

A PRINT statement can end with a comma, semicolon, colon, or no punctuation at all. A PRINT statement that ends with a semicolon causes the cursor to wait at the next position until another PRINT statement is executed. The cursor waits at the next available column when a PRINT statement ends with a comma. When a PRINT statement ends with a colon or has no punctuation at the end, the next line of output occurs on the next line. If more than one colon is used at the end of a statement, a line is skipped for each colon.

Example

```
> 100 X = 576
> 200 Y = 24
> 300 PRINT "CODE:"; X,
> 400 PRINT "UNITS:"; Y
> RUN
  CODE: 576    UNITS: 24
```

---

*In Extended BASIC, be sure to separate colons in a PRINT statement by one space.

In the previous example, the program contains two PRINT statements for one line of output. At lines 100 and 200, the variables are assigned values. At line 300, the string constant "CODE:" is printed, followed by the value of the variable X. Since a semicolon separates the values, they are printed adjacent to each other, separated by one space. The PRINT statement in line 300 ends with a comma, so the next output occurs in the second column of the same line. The string constant "UNITS:" is followed by the value of the variable Y. These values are also separated by one space.

A PRINT statement may include the optional file number. The output will be sent to the device that was opened with the specified filenumber. Filenumbers 1 through 255 are used for devices other that the screen. Filenumber zero is always open for output to the display.

Example

```
100 OPEN #1:"CS1",SEQUENTIAL,OUTPUT,FIXED
200 FOR J = 1 TO 10
300 X = RND
400 PRINT #1:X
500 NEXT J
600 CLOSE #1
```

The program in the previous example uses a PRINT statement to record ten random numbers on a cassette tape. At line 100, an I/O channel is opened for output to the cassette recorder. Line 200 indicates a FOR/NEXT loop that is repeated ten times. At line 300, a random value is assigned to the variable X. At line 400, the value of X is printed on the output device opened as file #1. When ten values are recorded, this file will be closed.

When a PRINT statement includes the REC option, the data will be written starting at the indicated *record number*. The REC option may only be used when writing onto a RELATIVE file.

PRINT USING

■ EX BASIC
□ ST BASIC

The PRINT USING statement is used to write string or numeric data in a predefined format.

Configuration

PRINT [#a][,REC *record number*] USING $\left\{\begin{matrix}\textit{line number}\\ \textit{expression}\end{matrix}\right\}$:*data list*

The file number and the record number are used the same way in the PRINT USING statement in the PRINT statement.

The expression or the line number following the USING statement defines the format in which the data will be written. If a line number is used, it must be the line number of an IMAGE statement. The expression used after the USING statement must contain special characters that are used to define the display format. These special characters can be found in Table 5-4 on page 194.

PRINT USING and IMAGE statements use the same procedure to define a display format.

Example

```
> PRINT USING "THE VALUE OF PI IS #.####":PI
  THE VALUE OF PI IS 3.1416
```

RANDOMIZE

■ EX BASIC
■ ST BASIC

The RANDOMIZE statement is used to reset the random number generator.

Configuration

RANDOMIZE [a]

If a seed is used with the RANDOMIZE statement, the same sequence of numbers will be generated each time the program is run. If the seed is omitted, an unpredictable sequence will be generated each time the program is run.

Example

```
> RANDOMIZE 25
```

The RND function is used to return the values calculated by the random number generator.

READ

- EX BASIC
- ST BASIC

A READ statement is used to assign values to variables. The values are taken individually from DATA statements in the order they appear in the program.

Configuration

$$\text{READ} \left\{ \begin{matrix} X \\ X\$ \end{matrix} \right\} \left[\begin{matrix} ,Y \\ ,Y\$ \end{matrix} \right] \ldots$$

Data items are assigned to variables in the order in which they appear in the program unless a RESTORE statement has been executed.

The type of variable in the READ statement must correspond to the type of data in the corresponding DATA statement. A numeric variable can only be assigned a numeric value. However, a string variable can accept any type of character or none at all.

A program must include at least as many data items as the number of variables in its READ statements unless a RESTORE statement is executed.

Example

```
> 20 READ X,X$
> 30 PRINT X$,X
> 40 END
> 50 DATA 12,JONES
> RUN
  JONES      12
```

The preceding example contains a program that has a READ statement. At line 20, the variables X and X$ are assigned the values from the DATA statement at line 50. At line 30, the values of the two variables are displayed.

A READ statement can accept data from a DATA statement that appears anywhere in a program. A DATA statement does not have to precede the READ statement in order to be effective.

REC

■ EX BASIC
□ ST BASIC

The REC function is used to determine the current record position in a RELATIVE file.

Configuration

X = REC (*filenumber*)

RELATIVE files are a method of storing information in which any record can be accessed within that file without first searching all of the preceding records.

Example

```
100 PRINT REC (1)
```

REM

■ EX BASIC
■ ST BASIC

A REM statement is used to insert comments in a program. The REM statement is ignored by the BASIC interpreter.

Configuration

REM *remarks*

Example

```
10 REM INPUT ROUTINE
```

Any statements that follow a REM statement on the same program line are also ignored by the computer. As a result, a REM statement is generally used on its own line or at the end of a mutilple statement line.

RESEQUENCE (RES)

■ EX BASIC
■ ST BASIC

The RESEQUENCE command is used to renumber the statements in a program.

Configuration

RESEQUENCE [*new#*][,*value*]

New# is the first new line number to be used in the renumbering process. The default value for *new#* is 100. *Value* is the amount to be added to each line number to generate the subseuqent line number. The default for *value* is 10.

The RESEQUENCE command also changes any reference to a line number that appears in any other statement. Line number references are updated in all of the following statements.

GOTO	BREAK	PRINT USING
GOSUB	RESTORE	DISPLAY USING
THEN	RETURN	ON ERROR
ELSE	UNBREAK	ON WARNING
RUN		

Example

```
> 1 X = X+1
> 2 PRINT X
> 3 GOTO 1
> RESEQUENCE 100, 10
> LIST
  100 X = X+1
  110 PRINT X
  120 GOTO 100
```

The previous example demonstrates the use of the RESEQUENCE command. Notice that the line numbers and the line number reference in the GOTO statement are all updated.

RESTORE

■ EX BASIC
■ ST BASIC

A RESTORE statement is used to move the data pointer.

Configuration

RESTORE [*ln*]

The data in a program is read in order, starting with the first DATA statement item. In order to reread a section of data, a RESTORE statement is necessary.

When a RESTORE statement is executed without an argument, the next READ statement will assign to its first variable the first data value that appears in the program.

When a RESTORE statement is executed with an argument, the next READ statement will assign to its first variable the first data value that appears at the specified line number.

Example

```
> 100 DATA 1,2,3,4,5
> 200 DATA 6,7,8,9
> 300 READ A,B,C,D,E
> 400 PRINT A;B;C;D;E
> 500 RESTORE
> 600 READ F,G,H,I,J
> 700 READ K,L,M,N
> 800 PRINT F;G;H;I;J;K;L;M;N
> RUN
   1 2 3 4 5
   1 2 3 4 5 6 7 8 9
```

In the preceding example, a RESTORE statement causes the same DATA statement to be read twice. If the RESTORE statement had not been included, there would not have been enough DATA for the READ statements in lines 600 and 700.

RESTORE *with files*

■ EX BASIC
■ ST BASIC

The RESTORE statement is used with files to move the pointer back to any position within a file.

Configuration

RESTORE *filenumber*, REC *recordnumber*

The *filenumber* indicates the file that is being manipulated. The *recordnumber* indicates the position within the file where the data pointer will be placed.

Example

RESTORE #1, REC 4

RETURN *with GOSUB*

■ EX BASIC
■ ST BASIC

A RETURN statement is used to branch a program back to the line where the last subroutine was called.

Configuration

RETURN

A subroutine is called with a GOSUB or ON/GOSUB statement. When the subroutine has been completed, a RETURN statement causes the program control to return to the statement following the most recently executed GOSUB or ON/GOSUB statement.

Example

10 RETURN

RETURN *with ON ERROR*

■ EX BASIC
□ ST BASIC

The RETURN statement is used with the ON ERROR statement to exit an error handling subroutine.

Configuration

RETURN { *ln* | NEXT }

The RETURN statement transfers program control to the main program after the execution of an error handling subroutine. RETURN can be used without an argument. In this case, program control will be transferred back to the line that caused the error.

Program control will be transferred to the line following the line that caused the error if the keyword NEXT is used with RETURN. The line that caused the error will not be executed again.

A line number may also be specified in the RETURN statement. In this case, program control is transferred to the line with the specified line number.

Example

```
100 ON ERROR 500
200 INPUT A
300 PRINT SQR(A)
400 GOTO 100
500 A = ABS(A)::RETURN
```

The preceding example contains a program that uses a RETURN statement without any options. When the program is executed, line 100 causes program control to be transferred to line 500 if an error occurs. Line 200 contains an INPUT statement that is used to assign a numeric value to the variable A. Line 300 prints the square root of the number that was input. Line 400 transfers program control back to the INPUT statement. Line 500 takes the absolute value of the number that caused the error.

Taking the square root of a negative number would normally cause an error. However, in this case, an error handling subroutine would take the absolute value of a negative number. The RETURN statement would then transfer program control back to line 300.

RND

■ EX BASIC
■ ST BASIC

The RND function is used to generate random numbers.

Configuration

X = RND

The computer uses a mathematical process to generate pseudo-random numbers greater than or equal to zero, but less than 1. The sequence of random numbers will be the same each time the computer is turned on unless a RANDOMIZE statement is used.

Example

```
100  PRINT INT (RND * 100)
200  GOTO 100
```

The previous example contains a program that will generate random number between 0 and 99 inclusive. The same numbers will be generated each time the program is executed unless a RANDOMIZE statement is inserted in the program. Press the FCTN 4 key (CLEAR) to stop the program.

RPT$

■ EX BASIC
□ ST BASIC

The RPT$ function is used to repeat a string a specified number of times.

Configuration

X$ = RPT$ (*string expression, numeric expression*)

The *string expression* is the expression that is to be repeated. The *numeric expression* determines the number of repetitions.

Examples

```
PRINT RPT$ ("A", 56)
PRINT RPT$ ("**", 14)
```

The previous example contains two statements. The first statement will cause the letter "A" to be printed 56 times on the screen. The second example will cause 28 asterisks (*) to be displayed on the screen.

More than one character may be used as the *string expression,* however, the length of the total expression after repetitions may not exceed 255 characters. If the generated string is more than 255 characters, a warning message will be displayed and the string will be truncated.

RUN

- EX BASIC
- ST BASIC

The RUN statement is used to execute the program that is currently in the computer's memory.

Configuration (EX & ST BASIC)

RUN [*ln*]

The RUN statement can include a line number as an argument. If the specified line number appears in the program, execution begins with that line. If the specified line number does not appear in the program, the following message will be displayed:

* BAD LINE NUMBER

If the RUN statement does not include an argument, execution will begin at the first line of the program.

Examples

```
RUN
RUN 100
```

The RUN command can be used in Extended BASIC the same way it is used in Standard BASIC. However, Extended BASIC allows more options to be used with the RUN command.

Configuration (EX BASIC)

RUN [*In* / *"device name.program name"*]

A RUN statement with no argument and a RUN statement with a line number have the same effect in Extended BASIC as they do in Standard BASIC.

In Extended BASIC, the RUN command can be used with an input device to load and execute a program. The contents of the computer's memory will be erased before the new program is entered into memory.

RUN can also be used as a statement in a program. This means that one program can load and execute another program.

Examples

```
RUN "CS1"
RUN "DSK1.TEST"
100 RUN "DSK1.TEST"
```

The preceding example contains three statements. The first statement causes the first program on cassette drive 1 to be loaded into memory and executed.

The second statement causes the program named TEST on disk drive 1 to be loaded into memory and executed. The third statement is an example of a RUN statement that is used in a program.

SAVE

- EX BASIC
- ST BASIC

The SAVE command is used to record the program that is currently in the computer's memory.

Configuration (EX & ST BASIC)

SAVE *devicename* . *programname*

The *devicename* specifies the device that will be used to save the program. Proper device names include CS1, CS2, DSK1, DSK2, and DSK3.

The *programname* used to specify disk program files can contain up to ten characters. Periods and spaces are the only characters that cannot be used in a program name.

For best results, however, use only upper case letters, numbers and the following symbols when specifying program names.

"#$%&'()*+,—/:;=?@[ ]!<>\^_

When using the SAVE command with a cassette recorder, a *programname* is not used.

Example

SAVE DSK1.PROGRAM1

The previous example contains a SAVE command that is used to record the program currently in the computer's memory. The program is saved on the diskette in disk drive 1 with the name PROGRAM1. There must be a program in the computer's memory before the SAVE command can be used.

When the SAVE command is used with a cassette recorder, a set of instructions will appear on the screen. These step by step instructions guide the user through the process of using the cassette recorder.

Example

```
>SAVE CS1

 * REWIND CASSETTE TAPE      CS1
   THEN PRESS ENTER

 * PRESS CASSETTE RECORD     CS1
   THEN PRESS ENTER

 * RECORDING

 * PRESS CASSETTE STOP       CS1
   THEN PRESS ENTER

 * CHECK TAPE (Y OR N)?      N
```

The previous example contains the results of a SAVE command being used with a cassette recorder.

The preceding example contains the prompts that normally appear when the SAVE command is executed. The prompts direct the user in the operation of the cassette recorder.

The amount of time required to record a program depends upon the length of the program.

When the computer completes the recording, it once again prompts the user on the operation of the cassette recorder. At this point, the user has the option of checking the tape. A recorded program should generally be checked to be sure that no errors developed while recording the program.

In the preceding example, the check option was not chosen. If "Y" had been entered in response to the CHECK TAPE prompt, a series of prompts would have appeared to direct the user in the operation of the cassette recorder.

If an error is detected while checking the tape, an error message will be displayed. At this point, the user has three options.

Example

```
* ERROR — NO DATA FOUND
  PRESS R TO RECORD
  PRESS C TO CHECK
  PRESS E TO EXIT
```

The preceding example shows an error message that may occur during the checking procedure. Three options are provided. If "R" is pressed, the computer will record the program again. If "C" is pressed, the computer will recheck the program on cassette. If "E" is pressed, the computer will abort the SAVE command.

In Extended BASIC, there are two additional options that can be used with the SAVE command. These two options are MERGE and PROTECTED.

Configuration (EX BASIC)

SAVE *devicename . program name* [,PROTECTED / MERGE]

If the keyword PROTECTED is used at the end of the SAVE command, the program will be saved in a special format. A program that was protected when it was saved cannot be listed, edited or resaved after it has been reloaded into memory.

If the keyword MERGE is used at the end of the SAVE command, the program can only be retrieved using the MERGE command. The MERGE option allows a program to be loaded into memory from a disk program file without erasing the program already in memory. The MERGE option can only be used with a disk program file.

SEG$

- EX BASIC
- ST BASIC

The SEG$ statement is used to designate characters in the middle of a string.

Configuration

A$ = SEG$(B$,*a*,*b*)

The SEG$ function returns a string value. The first argument is a string constant or a string variable. The second and third arguments are numeric values. The first numeric argument determines the first character from the string argument that is returned. The second numeric argument determines the total number of characters that are returned.

Example

```
> 10  A$ = "JOHN PETER JONES"
> 20  PRINT SEG$ (A$,6,5)
> 30  END
> RUN
  PETER
```

The previous example contains a program that uses the SEG$ function. At line 10, the variable A$ is assigned a string value. At line 20, the PRINT statement includes a SEG$ statement. The SEG$ statement specifies the string value of the variable A$. The second argument (6) specifies the sixth character in the string. The third argument (5) specifies the number of characters returned. As a result, the string value "PETER" is printed because "P" is the sixth character of the string (including spaces).

SGN

- EX BASIC
- ST BASIC

The SGN function returns a +1 if its argument is positive, a –1 if negative, and a 0 if zero.

Configuration

SGN (a)

Example

```
> 100  A = 100
> 200  X = SGN (A)
> 300  PRINT X
> RUN
   1
```

SIN

- EX BASIC
- ST BASIC

The SIN function returns the sine of the angle specified by its argument. The argument must be an angle measured in radians.

Configuration

X = SIN (a)

Example

```
> PRINT SIN (3.1415927/2)
  1
```

SIZE

■ EX BASIC
□ ST BASIC

The SIZE command returns the number of bytes of memory available.

Configuration

SIZE

Example

```
> SIZE
  13928 BYTES FREE
```

The previous example demonstrates the use of the SIZE command.

The size command displays additional information when a memory expansion unit is in use.

SQR

■ EX BASIC
■ ST BASIC

The SQR function returns the positive square root of its argument.

Configuration

X = SQR (a)

Example

```
> 100  X = 49
> 200  PRINT SQR (X)
> RUN
   7
```

STOP

■ EX BASIC
■ ST BASIC

The STOP statement causes a halt in the execution of a BASIC program.

Configuration

STOP

Generally, the STOP statement and the END statement are interchangeable. However, a STOP statement cannot be used at the end of a subprogram.

STR$

■ EX BASIC
■ ST BASIC

The STR$ function returns the string representation of its argument.

Configuration

X$ = STR$ (a)

The argument of an STR$ function must be a numeric value.

A numeric argument that has been converted to a string can no longer be used in any calculations. However, the string representation consists of the same characters as the original numeric argument.

Example

```
> 100  A$ = STR$ (40)
> 200  PRINT A$
> RUN
  40
```

SUB

■ EX BASIC
□ ST BASIC

The SUB statement is used as the opening statement of a subprogram.

Configuration

SUB *subprogram name* [(*variable list*)]

User written subprograms must begin with a SUB statement, which indicates the name of the subprogram. A subprogram must end with a SUBEND statement.

A CALL statement is used to transfer program control to a subprogram. For example, the statement CALL WORK would transfer program control to the subprogram named WORK. Once a SUBEND statement is encountered, program control is transferred back to the program line that follows the CALL statement that called the subprogram.

Subprograms must appear at the end of the main program. If more than one subprogram exists in a program, the subprograms

must be listed one after the other at the end of the main program.

Variables within the main program are independent of the variables within subprograms. For example, if the variable A is assigned the value of 10 within the main program, the value of A within a subprogram is zero until it is assigned a value. Variables which have been assigned values in the main program can be used within subprograms by including a variable list after the subprogram name. A variable list must appear in the CALL statement as well as the SUB statement.

When values are used in a subprogram, they must be passed to the subroutine variables. When a subroutine is called, the specified values are transferred from the CALL statement to the variables in the SUB statement. Illustration 5-7 depicts the passing of values.

Illustration 5-7. Passing Values to a Subprogram

```
CALL FUNCTION (X, Y, Z ...)
               ↓ ↓ ↓
SUB FUNCTION (A, B, C ...)
```

Example

```
> 100 X = 25
> 200 CALL MESSAGE (X)
> 300 PRINT SQR (X)
> 400 SUB MESSAGE (A)
> 500 PRINT "THE SQUARE ROOT OF";A;"IS";
> 600 SUBEND
> RUN
   THE SQUARE ROOT OF 25 IS 5
```

In the preceding example, a subprogram is used to print a message. Line 100 assigns the value of 25 to the variable X. Line 200 transfers program control to the subprogram MESSAGE. Line 300 prints the square root of X after the subprogram has been completed.

Line 400 is the beginning of the subprogram MESSAGE. Line 500 prints the message and line 600 returns program control to line 300.

A variable list is used in the CALL statement and SUB statement in order to transfer the value of X into the subprogram. In the subprogram, A is assigned the value of the variable X, which is 25.

SUBEND

■ EX BASIC
□ ST BASIC

The SUBEND statement is used to mark the end of a subprogram.

Configuration

SUBEND

Subprograms that begin with the SUB statement must end with a SUBEND statement. Once the SUBEND statement is executed, program control will be transferred to the line following the CALL statement that called the subprogram.

Example

```
> 100 FOR T = 1 TO 3
> 200 CALL FUNCTION (T)
> 300 NEXT T
> 400 SUB FUNCTION (A)
> 500 PRINT A^2,
> 600 PRINT A^3
> 700 SUBEND
> RUN
  1              1
  4              8
  9              27
```

The program in the preceding example contains a subprogram named FUNCTION. The subprogram displays the square and the cube of the argument.

When the SUBEND statement is executed, the program control branches to the statement that follows the CALL statement.

SUBEXIT

■ EX BASIC
□ ST BASIC

The SUBEXIT statement is used to exit a subprogram before a SUBEND statement is executed.

Configuration

SUBEXIT

The SUBEXIT statement is optional within a subprogram. The SUBEXIT statements can be used to branch out of a subprogram back to the main program.

Once a SUBEXIT statement is executed, program control will be transferred to the line following the CALL statement that called the subprogram.

Example

```
100  INPUT X
200  CALL CHECK (X)
300  SUB CHECK (A)
400  IF A >0 THEN PRINT" AMOUNT:";A::SUBEXIT
500  PRINT "INVALID AMOUNT"
600  SUBEND
```

The preceding example demonstrates the use of a SUBEXIT statement. At line 100, a numeric value is input and assigned to the variable X. At line 200, a subroutine is called.

The subroutine is passed the value of the variable X. As a result, wherever the variable A appears in the subroutine, the value of the variable X will be used.

The subroutine prints one of two messages, depending on the value of the variable A. If the value is positive, the statements at line 400 will be executed and the subroutine will be exited.

If the value of the variable is not a positive value, the statement at line 500 is executed. When a SUBEND statement is executed, program control branches to the statement that follows the CALL statement.

TAB

■ EX BASIC
■ ST BASIC

The TAB statement is used to specify the column where the next item in a PRINT or DISPLAY statement will be output.

Configuration

TAB (a)

The output that follows a TAB statement begins at the column specified by the argument.

Example

```
PRINT X$; TAB (8); Y$
```

In the previous example, the value of the variable X$ is displayed, starting in the first column. The value of the variable Y$ is displayed, starting in column number 8.

If the next available column is greater than the argument of a TAB statement, the data following the TAB statement will be printed on the next line starting at the specified column.

TAN

■ EX BASIC
■ ST BASIC

The TAN function returns the tangent of the angle specified as its argument. The argument will be assumed in radians.

Configurations

X = TAN (*a*)

Example

```
> PRINT TAN (3.14159265359)
  0
```

TRACE

■ EX BASIC
■ ST BASIC

The TRACE command is used to follow the execution of program statements.

Configuration

TRACE

The TRACE command sets a trace flag that causes the line number of each statement in the program to be printed as it is executed. The line numbers will be displayed within angle brackets.

Example

```
> 100 X = 100
> 200 FOR I = 1 TO 3
> 300 X = X/2
> 400 PRINT X
> 500 NEXT I
> 600 END
> TRACE
```

```
> RUN
  < 100 >  < 200 >  < 300 >  < 400 >     50
  < 500 >  < 300 >  < 400 >    25
  < 500 >  < 300 >  < 400 >    12.5
  < 500 >  < 600 >
```

UNBREAK

■ EX BASIC
■ ST BASIC

The UNBREAK command is used to remove breakpoints that were inserted in a program.

Configuration

UNBREAK [*ln*][,*ln*] ...

UNBREAK commands can be used to eliminate breakpoints that are specified by line number only. For example, if a program includes the following statement,

```
100  BREAK 200,250,260
```

an UNBREAK statement can eliminate all of these breakpoints. In this case, the UNBREAK statement must occur in the program after the BREAK statement.

If the following BREAK statements appear in the program, an UNBREAK statement will have no effect.

```
200 BREAK
250 BREAK
260 BREAK
```

When the breakpoints are specified by a command that is not part of the program, the breakpoints can be easily eliminated. For example, an UNBREAK statement can be used counteract the following BREAK statement.

BREAK 200,250,260

Examples

```
110  UNBREAK
110 UNBREAK 200,250,260
UNBREAK
```

UNTRACE

■ EX BASIC
■ ST BASIC

The UNTRACE command cancels the effect of the TRACE command.

Configuration

UNTRACE

Example

```
> 100 X = 10
> 110 PRINT X
> 120 Y = 20
> 130 PRINT Y
> TRACE

> RUN
      < 100 >  < 110  > 10
      < 120 >  < 130  > 20

    **  READY  **
> UNTRACE

> RUN
  10
  20
```

VAL

■ EX BASIC
■ ST BASIC

The VAL function converts its string argument to a numeric value. The numeric characters in the string argument will be converted to their numeric equivalents. Non-numeric characters may not be used in the argument.

Configuration

X = VAL (A$)

Example

```
> 100 A$ = "57342"
> 200 PRINT VAL (A$)
> 300 PRINT VAL (A$) + 2
> RUN
  57342
  57344
```

XOR

■ EX BASIC
□ ST BASIC

XOR is used between two expressions as either a numeric or logical operator.

Configuration

ex XOR ex

The conditions of true and false are represented in the computer by the logical values -1 and 0. As a result, the logical operators (AND, OR, XOR and NOT) operate with the logical values -1 and 0. The XOR operation can be explained by the following truth table.

EX1	EX2	RESULT
-1	-1	0
-1	0	-1
0	-1	-1
0	0	0

The XOR logical operator is generally used in an IF,THEN statement with relational expressions.

Example

```
10 INPUT X
20 INPUT Y
30 IF X < O XOR Y < O THEN 50
40 END
50 PRINT X:"TIMES";Y;"IS NOT POSITIVE"
```

The preceding example demonstrates the use of the XOR logical operator. Lines 10 and 20 are used to assign numeric values to the variables X and Y. The XOR operator is used in line 30 to determine if either (but not both) of the arguments are negative. If this is true, the PRINT statement at line 50 will be executed. If X and Y are both positive or both negative, the program will have no output.

Typical output for the example program is as follows.

```
-1 TIMES 5 IS NOT POSITIVE
```

When XOR is used as a numeric operator, the arguments will be rounded off and converted to their binary equivalents. The value of the XOR statement will be the result of the XOR operation on each bit of the values. For example:

```
     10    (binary 1010)
 XOR 12    (binary 1100)
 ---------------------
      6    (binary 0110)
```

Since the binary equivalent of 10 is 1010, and the binary equivalent of 12 is 1100, the XOR operation for each bit has the result 0110. The decimal value of 0110 is 6.

CHAPTER 6.
THE TI PROGRAM RECORDER

INTRODUCTION

The Texas Instrument Program Recorder is used for storing BASIC programs or data on cassette tape. The process of transferring a program from the computers memory onto cassette tape is known as **saving** a program. Once a program has been saved, it can later be transferred back from the storage device into the computer's memory. This process is known as **loading.**

Data can also be transferred back and forth between the computer and the Program Recorder. The process of saving data on cassette tape is known as **writing** the data. The retrieval of that data from the cassette tape is known as **reading** the data.

Installation

The installation of the TI Program Recorder is simple and straightforward. Two cords are supplied with the Program Recorder. One of the cords is a power cord that allows the Program Recorder to be used without batteries. The other cable is used to transmit signals between the computer and the cassette recorder.

The data cable has a single plug on one end, and 3 smaller plugs on the other. The large plug should be inserted in the receptacle on the back of the computer console. The three smaller plugs should be inserted in the side of the Program Recorder. The three jacks on the Program Recorder are marked with the color that corresponds to the appropriate plug (white, red or black).

Be sure to follow the complete set of installation instructions in the Program Recorder manual.

Saving & Loading Programs

The Program Recorder can be used to store the program that resides in the computer's memory. Many lengthy programs can be stored on a single cassette tape. However, if more than one program is stored on one side of a cassette tape, the position of each program must be noted. If programs are recorded haphazardly on a single cassette, it will become nearly impossible to preserve and recover them.

There are two BASIC commands that can be used to save and load programs. These commands are SAVE and OLD. The SAVE command is used when programs are to be stored, and the OLD command is used to recover a previously stored program.

SAVE

When the SAVE command is used to store a program, the command must include the device name CS1 or CS2. If only one Program Recorder is being used with the computer system, the device name CS1 must be used. However, a special cable may be purchased that allows two Program Recorders to be used. In this situation, either device name, CS1 or CS2, may be used.

The proper configuration for a SAVE command is as follows.

```
>SAVE "CS1"
```

The SAVE command can only be used if a program is currently stored in the computer's memory. If a program does not exist, a CAN'T DO THAT error will occur when the SAVE command is entered.

When the SAVE command is used properly, a series of instructions will be displayed on the screen to guide the user through the use of the the Program Recorder. These instructions are generally quite easy to understand.

When the SAVE command is entered, the first instruction is displayed as follows.

*REWIND CASSETTE TAPE
THEN PRESS ENTER

This instruction is a reminder to position the tape in the location at which you would like to record the program. It is important to choose a blank section of the tape for the program to be saved. If the tape is positioned where another program has been recorded, and the previously recorded program will be erased. The tape counter can be used to keep track of the precise location of the programs on a cassette tape.

When the tape has been properly positioned and the Enter key has been pressed, the following instruction will be displayed.

*PRESS CASSETTE RECORD
THEN PRESS ENTER

This instruction is a reminder to press the appropriate levers on the Program Recorder. When a program is to be saved, the RECORD and PLAY levers must pressed simultaneously. Both levers must be pressed firmly so they lock into position.

When the Enter key is pressed, the next message will be displayed as follows.

*RECORDING

This message indicates that the Program Recorder is operating and the program is being recorded. Intervals of sound may be heard while the data is being transferred. These sounds indicate that the data is actually being sent to the Program Recorder.

When the transfer of data is complete the final instruction will be displayed as follows.

*PRESS CASSETTE STOP THEN PRESS ENTER

This message indicates that the recording process is complete. The STOP lever on the Program Recorder should be pressed in order to release the PLAY and RECORD levers.

Checking Programs

The TI Program Recorder can be used to verify that a program was recorded properly. This feature is essentially a re-reading of the program to determine if the program was recorded in the proper format.

When the SAVE command is used, the CHECK TAPE option will be presented after the recording procedure has been completed. This option requires a simple Y or N response to the prompt. If N is typed, the recording will not be verified. If Y is typed, another series of instructions will be displayed.

The instructions used to verify the program are exactly the same as the instructions used with the OLD command. These instructions are completely described in the following section.

OLD

The OLD command is used to recover programs that were previously recorded. Whenever a program is loaded into the computer's memory by means of an OLD command, any program that may have previously been present in the computer's memory will be erased. The OLD command presents a list of instructions that are similar to the instructions of the SAVE command. The format for a typical OLD command is as follows.

```
>OLD "CS1"
```

When an OLD command is entered, the following instruction will appear on the display.

```
*REWIND CASSETTE TAPE
THEN PRESS ENTER
```

This message is a reminder to position the tape at the beginning of the desired program. When this step has been completed and the ENTER key has been pressed, the second message will be displayed as follows.

*PRESS CASSETTE PLAY
THEN PRESS ENTER

This message indicates that the PLAY lever on the Program Recorder should be pressed. Be sure that the lever is pressed down completely and that it locks into position. Proceed by pressing the Enter key.

While the transfer of data takes place, tones may be heard from the television or monitor speaker. While this procedure is performed, the following message will be displayed.

*READING

If no problems occur during the loading of the program, the following two messages will be displayed.

*DATA OK
*PRESS CASSETTE STOP
THEN PRESS ENTER

This final message reminds the user to press the STOP lever on the Program Recorder in order to release the PLAY lever. When this final step has been completed, the computer will be ready to accept another command.

Errors

There are generlly two types of errors that occur with the Program Recorder. These two errors are NO DATA FOUND and ERROR DETECTED IN DATA. These errors generally occur due to incorrect volume setting on the Program Recorder. If one of these errors occurs frequently, refer to the Program Recorder manual for troubleshooting hints.

When an error occurs, the following list of selections will be displayed on the screen.

PRESS R TO READ
PRESS C TO CHECK
PRESS E TO EXIT

If selection R is chosen, the computer will repeat the procedure that is used to load a program. If selection C is chosed, the computer will execute the procedure that checks the condition of the recorded program. If selection E is chosen, the computer will cancel the OLD command and will be prepared to accept another command.

Saving Data

The TI Program Recorder is not restricted to saving and loading programs. It can also be used to save data. Any string or numeric values that need to be saved can be recorded and retrieved with the Program Recorder.

There are 4 BASIC commands that are used to store and retrieve data. These statements are OPEN, CLOSE, PRINT# and INPUT#.

OPEN

Before data can be output to any external device, an OPEN statement must be executed. The OPEN statement must include the information that is required to establish communication between the computer and the external device.

A filenumber is a parameter that is used to specify a particular channel of communication between the computer and a particular device. It is necessary to use a filenumber because several different peripheral devices may be in use at the same time, or the same device may be used for several functions simultaneously. To avoid confusion, each input or output operation of the computer must be assigned a unique number.

The filenumber must be an integer from 1 to 128, and must be the first parameter in an OPEN statement. Any subsequent PRINT#, INPUT# or CLOSE statement must include the specified filenumber.

An OPEN statement must also include several additional parameters to specify the exact nature of the transfer of data. The most important parameter is either INPUT or OUTPUT. This parameter indicates whether the computer is accepting information (input) or sending information (output).

In general, the keywords INTERNAL and FIXED must in included in an OPEN statement for the Program Recorder. As a result, a typical OPEN statement for the Program Recorder would appear as follows.

```
OPEN #1:"CS1",OUTPUT,FIXED,INTERNAL
```

CLOSE

A CLOSE statement is used to eliminate a channel that was previously established for an input or output operation. A CLOSE statement does not require any parameters other than the filenumber.

A CLOSE statement is not always required since each input or output channel is automatically closed when the program ends. However, it is a good programming practice to close each channel that is opened in a program.

A typical CLOSE statement would have the following structure.

```
CLOSE #1
```

PRINT#

A PRINT# statement is used to output data to the Program Recorder. The format for this statement is basically the same as the PRINT statement, except for the filenumber, followed by a colon. The values that follow the colon are output to the specified device.

For example, the following PRINT# statement would cause the values of the variables A$,B$ and C$ to be output to the Program Recorder.

```
PRINT#1:A$,B$,C$
```

When data is output to the Program Recorder, the data items in the PRINT# statement should be separated by commas.

The data is output to the Program Recorder is sections called **fields.** A field is merely a segment of the tape that is used to store one or more data items. Each PRINT# statement outputs exactly one field.

For the Program Recorder, each field is 64 characters long unless a smaller number is specified in the corresponding OPEN statement. As a result, the data that is sent to the Program Recorder cannot exceed a total of 64 characters.

The length of a string value is the total number of characters in the string. Letters, numbers spaces and punctuation marks all contribute to the length of a string value. Each numeric value has the equivalent length of an 8 character string value.

One space of the data field must be used to separate the individual values. These spaces must also be considered when field lengths are determined. As a result, the total number of characters in the field can be calculated as follows.

```
  Total number of variables
  Total number of characters in string values
+ Eight characters for each numeric value
-------------------------------------------
  Total field lengths
```

For example, consider the program on the following page.

```
10  OPEN #1:"CS1",OUTPUT,INTERNAL,FIXED
20  A=53.5
30  B=47.85
40  A$="TEXAS INSTRUMENTS"
50  PRINT #1:A,B,A$
60  CLOSE #1
```

The preceding example program outputs the values of three variables to the Program Recorder. Line 10 is an OPEN statement that reserves I/O channel number 1 for output to the Program Recorder. Lines 20 through 40 assign values to the variables A,B and A$. At line 50, a PRINT# statement is used to output these values in a single field. The output of this example program can be recovered by the program on page 265.

When the sample program is executed, the prompts will appear on the display as a reminder of the correct use of the Program Recorder.

Since only one PRINT# Statement is executed in the program, the output consists of only one field. The length of the field can be calculated as follows.

```
  Total number of variables: 3
  Total number of characters in string values: 17
+ Eight characters for each numeric value: 16
  ---------------------------------------------
  Total field lengths: 36
```

Since the total field length is less than 64, the PRINT# statement has a valid format.

When a PRINT# statement is used to output a field that is too long, a FILE ERROR will occur. To prevent this problem, carefully examine each PRINT# statement and be sure that the length of each field does not exceed 64.

INPUT#

An INPUT# statement is used to retrieve data from the Program Recorder. The format for this statement is basicaly the same as the INPUT statement, except for the filenumber followed by a colon. The variables specified in the INPUT# statement are assigned values that have previously been stored in a cassette data file.

For example, the following INPUT# statement would cause values to be input for the variables X$,Y$ and Z$.

```
INPUT #1:X$,Y$,Z$
```

The variables specified in an INPUT# statement must be separated by commas.

Data is stored in a cassette data file as a collection of fields. Each field in the file has a specific format. The type of data in each field is determined by the structure of the PRINT# statement that was used to output the data. For example, if a PRINT# statement outputs two string values followed by two numeric values, these values are recorded in a specific order in a specific data field. As a result, the data in the field can only be recovered by an INPUT# statement that has the same structure. This principle is demonstrated in Illustration 6-1.

Illustration 6-1. Corresponding PRINT#/INPUT# Formats

Output Statement	Corresponding Input Statement
PRINT #1:A$	INPUT #4:X$
PRINT #27:A$,B$,C,D	INPUT #3:L$,M$,N,O
PRINT #1:A,B,C,D	INPUT #1:A,B,C,D
PRINT #4:A,B$,C	INPUT #6:X,X$,Y

There are several important considerations in the use of PRINT# and INPUT# statements. The most important detail is the fact that each corresponding pair of PRINT# and INPUT# statements must have the same number of variables. String variables must corres-

pond to string values and numeric variables must correspond to numeric values.

The actual names of the variables do not have to be exactly the same in a corresponding pair of statements. The only requirement is that the types of variables must be consistent.

The filenumbers in corresponding statements do not have to be the same. This is a result of the fact that the TI Program Recorder cannot be used for input and output simultaneously. As a result, the I/O channel that was used for output must be closed before a separate channel is opened for input. As a result, the two separate channels may or may not use the same filenumber.

The following example progran can be used to recover the data that was stored by the previous example program.

```
100  OPEN #1:"CS1",INPUT,INTERNAL, FIXED
110  INPUT #1:X,Y,Z$
120  PRINT X,Y,Z$
130  CLOSE #1
140  END
```

Notice that the INPUT# statement in this program has the same numbers and type at variables as the PRINT# statement in the previous example. These two programs can be combined to perform the input and output functions in a single program.

Whenever cassette data files are used, be sure to note the location of the beginning of the file. When the data is recovered from the file, the tape must be rewound to the beginning of the file. If this procedure is performed correctly, the output of the sample program should appear as follows.

```
 53.5                    47.85
TEXAS INSTRUMENTS
```

Files

Data files usually consist of many fields. Unfortunately, it becomes quite difficult to include a PRINT# statement in a program each time data is read or written. As a result, it is usually more practical to use a FOR/NEXT loop to repeat a section of a program. The following program demonstrates this principle.

```
 10  OPEN #1:"CS1",OUTPUT,INTERNAL, FIXED
 20  INPUT "NUMBER OF ENTRIES":N
 30  FOR J=1 TO N
 40  INPUT "DESCRIPTION:":D$
 50  INPUT "PRICE:":P
 60  INPUT "QUANTITY:":Q
 70  PRINT #1:D$,P,Q
 80  NEXT J
 90  PRINT #1:"END",0,0
100  CLOSE #1
```

The FOR/NEXT loop in the preceding program allows any number of fields to be entered in the data file. A field with the values "END",0,0 is entered as the last field so the end of the data can be recognized when the data is recovered.

The following example program can be used to recover the data that is stored in the data file.

```
 200  OPEN #1:"CS1",INPUT,INTERNAL,FIXED
 210  PRINT "INVENTORY REPORT"::
 220  INPUT #1:A$,B,C
 230  IF A$="END"THEN 999
 240  PRINT A$;B,C
 250  GOTO 220
 999  CLOSE #1
1000  END
```

The preceding example program uses an IF, THEN statement to test the data being input. If the input is equal to a specific value ("END" in this case) the program is ended. The GOTO statement

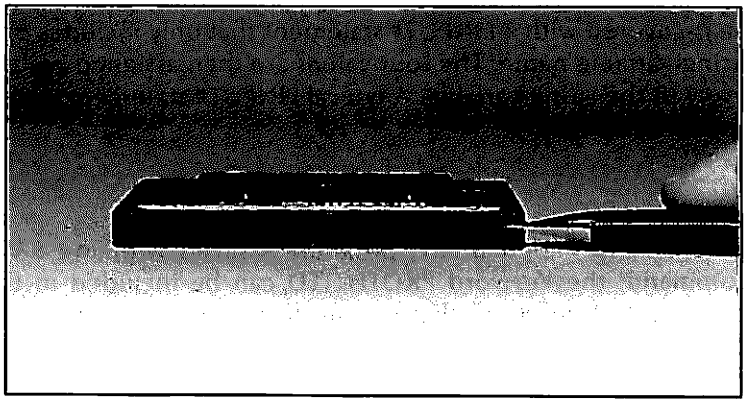
at line 250 causes the INPUT# statement to be repeated until the end of the file is detected.

Protecting Programs and Data

Programs and data that are stored on cassettes can be protected from accidental erasure. Each cassette tape cartridge has two tabs that are used to prevent the contents of the tape from being erased. When these tabs are removed, the Program Recorder will not be able to be used in the record mode. As a result important programs or data cannot be erased by accidentally recording other information on the same tape.

Illustration 6-2 depicts the technique used to "write protect" one side of a cassette tape.

Illustration 6-2. Write Protecting a Cassette Tape



You can determine which write protect notch protects which side of the tape by holding the cassette so that the exposed tape is facing towards you, and the side that is to be protected is facing up. By removing the tab on the left side of the cassette, the side of the tape facing up will be protected. By removing the tab on the right side, the side of the cassette facing down will be protected.

Once a cassette tape has been write protected, the effect can be reversed by covering the write protect notch with a small piece of tape.

Extended BASIC Features

When TI Extended BASIC is used with the TI-99/4A computer system, the Program Recorder takes on additional features. The most useful feature is that of protected programs.

Data can be read from a file in Extended BASIC with a LINPUT# statement. This technique allows all of the data stored in a record to be assigned to a single string variable. However, this statement can only be used with files that are stored in the DISPLAY mode rather than INTERNAL.

For example, a PRINT# statement may be used to write the values of four variables in a single cassette record. These four values can be recovered with a LINPUT# statement that only includes one string variable name. The four values are then assigned collectively to the string variable. A typical LINPUT# statement would appear as follows.

```
LINPUT#1:A$
```

When an Extended BASIC program is saved with the SAVE command, the keyword PROTECTED can be included in the command. A program that was protected when it was saved cannot be listed, edited or resaved after if has been reloaded into memory. This feature allows programs to be used repeatedly, but prevents them from being altered or copied. The following SAVE command demonstrates the format used to produce a protected version of a program.

```
>SAVE "CS1",PROTECTED
```

Entended BASIC also allows programs to be loaded and excuted with a single statement. The RUN command is used to perform this function. When a RUN command is executed, the list of

instructions will be displayed in the same manner as the OLD command. However, the program will be automatically executed when the loading procedure has been completed. This feature allows a RUN statement to appear in a program. As a result, a program that is being executed can be used to load and execute another program that is saved on a cassette tape. An example of this concept is located in the last section of Chapter 3. The correct system for a RUN command is illustrated by the following example statement.

```
>RUN "CS1"
```

CHAPTER 7. THE TI 1250 & 1850 DISK DRIVES

The Texas Instruments 1250 and 1850 disk drives provide a fast and efficient means of storing programs and data. Although disk drives cost more than cassette Program Recorders, disk drives are much more versatile and easier to use.

The two models of disk drives are basically the same. However, the 1250 disk drive is intended to be installed in a Peripheral Expansion box. The 1850 disk drive is built into its own enclosure, and is intended to be used separately.

The Disk Memory System is a package that includes a Disk Drive Controller and a Disk Manager Command Module. The Disk Drive Controller is an electronic device that actually manipulates the disk drives. This system is capable of controlling 1, 2, or 3 independent disk drives. No disk drive can be used with the computer system unless the Disk Drive Controller is installed.

The Disk Manager Command Module plugs directly into the computer console. This device allows the disk drives to be used to perform special functions. This Command Module does not need to be used during the normal operation of the disk drive. However, some special features of the disk drives require that the Command Module be inserted in the computer console.

A complete guide to the installation of the disk drives and Disk Drive Controller can be found in their respective manuals. Be sure to follow these instructions carefully.

Before a disk drive is actually used with the computer system, it is helpful to understand the fundamentals of disk storage.

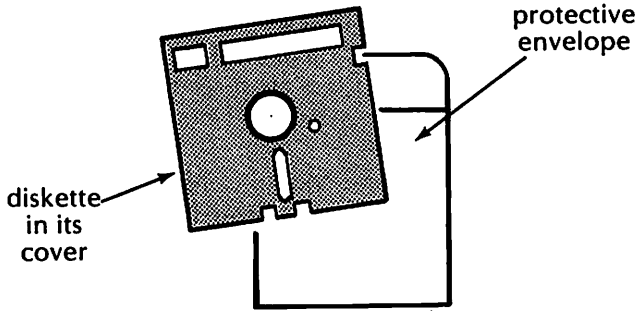
Floppy Diskettes

The most widely used type of disk storage with microcomputers is floppy disk storage. A floppy diskette consists of a round vinyl disk enclosed within a plastic cover. The diskette is generally stored in a diskette envelope.

This cover protects the diskette from damage while it is being handled by the operator. The diskette should never be removed from its cover. A 5¼ inch diskette with its protective envelope is shown in Illustration 7-1.

The diskette is allowed to rotate within the protective cover. The round hole in the middle of the diskette allows the disk drive to hold the diskette and spin it. The oblong shaped opening on the protective cover provides an area where data can be read from or written to the diskette surface.

Illustration 7-1. Mini-Floppy Diskette



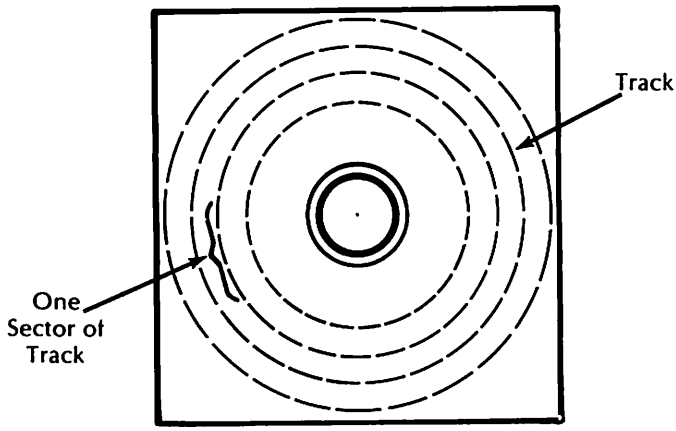
Tracks and Sectors

To facilitate the process of searching for data on the diskette surface, the surface is divided into tracks and sectors.

Tracks may be visualized as a series of concentric circles on the diskette surface, as shown in Illustration 7-2. There are 40 tracks on a diskette used by the TI disk drives.

To further reduce the time necessary to search for a particular data item, the tracks are divided into sectors, which are also shown in Illustration 7-2.

Illustration 7-2. Tracks and Sectors



Locating Tracks and Sectors

Locating a particular track on the disk surface is a relatively uncomplicated matter. The drive merely moves the head to the position on the diskette where the specified track is located, much like the needle on a phonograph is positioned to the location of a specific song on a record album.

However, locating a particular sector is a more difficult process. An index hole is used to determine the position of the diskette. It is located just to the right of the large hole in the middle of the 5¼ inch diskette.

The index hole, as shown in Illustration 7-1, is a hole only in the diskette's protective covering. Another index hole is located on the actual diskette surface inside the cover. As the diskette spins, the index hole on the diskette surface passes underneath the hole in the protective cover.

A light source inside the disk drive shines light onto the area of the diskette containing the index hole. When an index hole on the disk surface is aligned with the index hole on the protective cover the light will shine through to a sensor. The sensor will relay information on the location of the index hole, which can be used to calculate the various sector locations.

This method of locating sectors is called soft sectoring. Although the TI disk drives use soft sectored diskettes, some computers use a similar system called hard sectoring. Hard sectored diskettes have more than one index hole.

Single and Double Sided Diskettes

Some floppy diskettes are designed to be written on only one side. These are known as single sided (SS) diskettes.

Diskettes which are designed to be written on both sides are known as double sided (DS) diskettes.

Single, Double, and Quad Density Diskettes

Density refers to a diskette's recording format, which in turn affects its capacity. Single density 5¼ inch diskettes have roughly 95K of capacity, double density 5¼ inch diskettes have a capacity of about 150-200K, and quad density 5¼ inch diskettes have a capacity of up to 370K.

The TI disk drives use single sided, single density or double density diskettes. A small amount of the storage capacity of each diskette is reserved for the operations of the disk drives. The available storage capacity of each diskette is 90K.

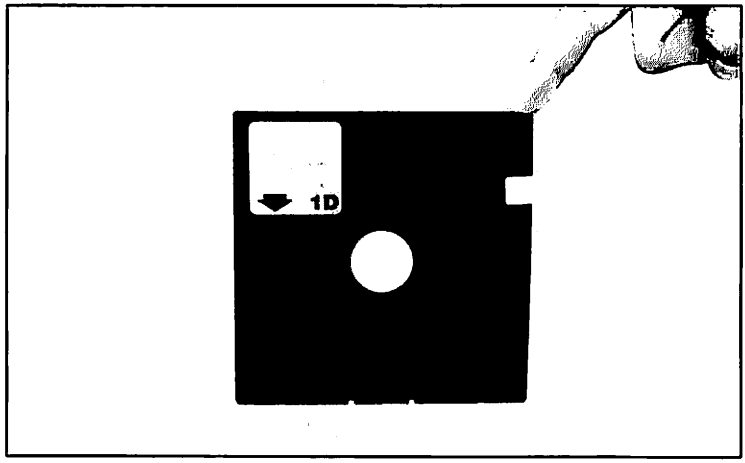
Diskette Write Protection

Diskettes have a notch on the side of their protective envelope that determines whether or not data can be written onto that diskette. On 8 inch diskettes, this notch is known as a write-protect notch. On 5¼ inch diskettes, it is known as a write-enable notch.

On an 8 inch diskette, information cannot be written onto the diskette unless this notch has been covered. On 5¼ inch diskettes, information cannot be written onto the diskette unless the notch is left uncovered.

Some 5¼ inch diskettes may be permanently write protected if their protective envelope does not contain a notch. Any 5¼ inch diskette with a notch can be write protected by merely covering the notch with a piece of tape as shown in Illustration 7-3.

Illustration 7-3. Write Protecting a 5¼ Inch Diskette



The main portion of this chapter is dedicated to the use of a single disk drive with the TI-99/4A computer system. However, all of this explanation applies to the use of several disk drives as well.

Powering On

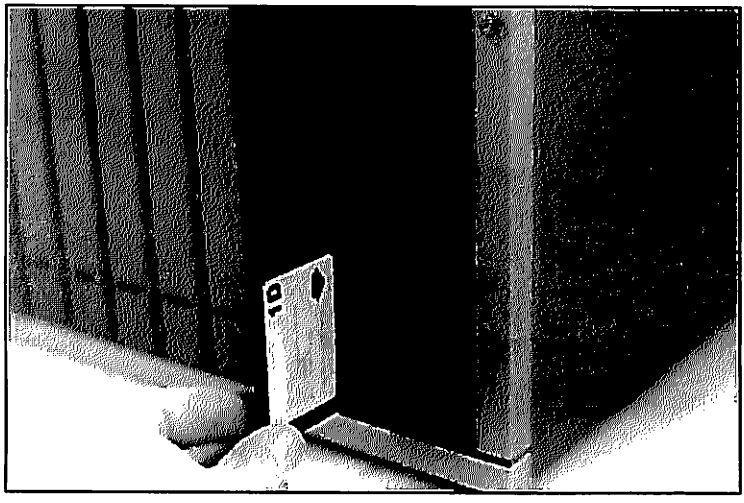
In order for a disk drive to be used with the computer system, the Disk Controller must be powered on before the computer console is powered on. Generally, the best technique is to power on all of the peripheral equipment, then power on the computer console last.

Inserting a Diskette

Before a diskette can be inserted in a disk drive, the drive door must be open. If the drive door is not open already, press the rectangular button to release the door latch. Do not attempt to force the door open, it should spring open automatically.

Insert a diskette into the disk drive as pictured in Illustration 7-4.

Illustration 7-4. Inserting a Diskette



Notice that the write-enable notch is at the top of the diskette, and the head access slot is forward. Once a diskette has been inserted in the drive, close the drive door by sliding it to the left. If the diskette interferes with the closing of the door, the diskette has not been inserted properly.

To remove a diskette, simply press the door latch release button. This causes the drive door to open, and the diskette to be partially ejected.

File Storage

When programs or data are output to a disk file, the file must be given a name. Valid file names must contain 10 characters or less,

and consist of uppercase characters and numbers. The following special symbols can also be used in a file name.

! # $ % & ' ( ) * + - / : ; < = > ? @ [/] ^ _

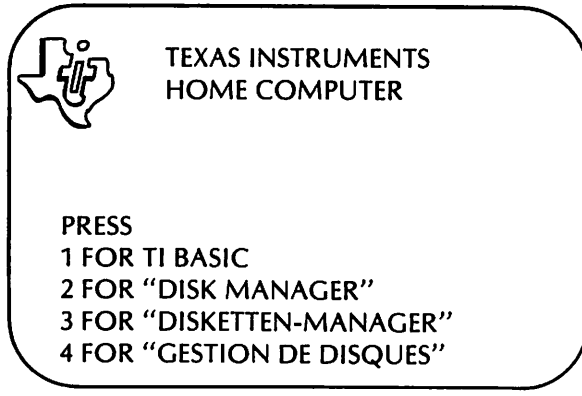
File names are used to distinguish the files that are recorded on a diskette. As a result, two files cannot exist on the same diskette with the same file name. If a file is saved with a file name that already exists on a specific diskette, the file that was recorded earlier will be erased and the new file will be saved.

Diskette files belong to two broad catagories: programs and data. Program files can contain only one BASIC program and data files can contain only data. Data and programs cannot be combined in a single file. The way in which these files are created is discussed in a later section of this chapter. This topic is deferred until the way in which the files are manipulated is discussed.

THE DISK MANAGER

The Disk Manager is a command module that is used to perform special functions of the disk memory system. These functions include initializing diskettes, deleting files and generating backup diskettes. The Disk Manager is easy to use because each function can be selected from a list of choices that are displayed on the screen. A list of selections of this type is called a **menu.**

When the Disk Manager Command module is inserted in the command module slot in the console, the master selection menu appears as follows.



Selection number 2 from this menu activates the disk manager. A title display appears for several seconds when this selection is chosen. Each menu that appears after this display requires that the proper selection be made and the proper data entered. The Enter key must be pressed after each response is entered.

Usually, when the computer requires a selection, a value appears at the location of the cursor. If the desired selection is the same as the value that is displayed, simply press the Enter key to make a selection. The values that automatically appear are called **default values.**

When a set of selections are chosen from a menu, the following prompt may appear at the bottom of the display.

PRESS: PROC'D, REDO,
BEGIN, OR BACK

The four commands correspond to the numbered keys (5, 6, 8 and 9) on the keyboard when they are used with the FCTN key. Be sure to hold the FCTN key down while striking the appropriate numbered key.

The PROC'D command causes the selected procedure to be performed. If all of the selections from the menu are correct, the PROC'D command is used to proceed with the specified function or repeat the previous function.

The REDO command causes the menu to be redisplayed so a different set of selections can be made. This command is usually chosen when a mistake is made in a menu selection.

The BEGIN command causes the main Disk Manager menu to be displayed. This command effectively returns the Disk Manager to its initial condition.

The BACK command causes the previous menu to be displayed. This command should be chosen when it is desirable to return to a previous level, but not all the way back to the main selection menu. The main selection list appears as follows.

DISK MANAGER

1 FILE COMMANDS
2 DISK COMMANDS
3 DISK TESTS
4 SET ALL COMMANDS FOR
SINGLE DISK PROCESSING

The most useful categories of commands are the file commands and disk commands. The file commands are used to manipulate the individual files on a diskette, and the disk commands deal with an entire diskette.

The disk tests commands are not as useful as the first two categories and they are rarely used. The last selection on the main Disk Manager menu is used to inform the Disk Manager that only one disk drive is being used with the computer system.

When the Disk Manager is in use, it is often necessary to specify the disk drive that contains the diskette for which the function should be performed. However, when the Single Disk Processing selection is chosen, it is no longer necessary to specify the desired disk drive.

File Commands

The individual file commands are named as follows.

COPY FILE
RENAME FILE
DELETE FILE
MODIFY FILE PROTECTION

As would be expected, the Copy File command is used to duplicate a program or data file. When this command is executed, the operator is asked to specify the disk drive that contains the master copy of the file to be copied. Next, the file name of the master file must be specified. The final two specifications that must be included are the disk drive number where the new copy should be located, and the file name that the newly copied file will have.

When all of the correct entries have been specified, press FCTN-6 (PROC'D) to begin the copying procedure. If an error was made in the specifications of the parameters, press FCTN-8 (REDO) to repeat the Copy File questionnaire.

The Single Disk Processing selection from the Disk Manager menu causes the Copy File procedure to be altered slightly. The Disk Manager assumes that only one disk drive is in use. As a result, Disk Drive number 1 is considered to be the master as well as the copy disk drive. To allow a single disk drive to be used to copy files on separate diskettes, the Single Disk Processing selection allows diskettes to be swapped during the copy procedure. A list of prompts are displayed to aid the operator through the disk swapping procedure. After each step, press FCTN-6 to continue the procedure.

The Rename File command is used to change the name of a program or data file. This command requires that the disk drive number, old file name and new file name be specified. In the Single Disk Processing mode, the disk drive number is assumed to be one. When all of the prompts have been given appropriate responses, press FCTN-6 to execute the command.

The Delete File command can be executed in either of two modes. When this command has been selected, the first prompt will appear as follows.

SELECTIVE (Y/N)?

If the response to this prompt was N, the disk drive number and the name of the file to be deleted must be specified. If the response to the prompt was Y, each file that is located on the specified disk drive will be listed by file name. Each file will be listed individually, and the following prompt will be displayed.

DELETE (Y/N)?

When this prompt appears, the operator will have the opportunity to either delete the listed files or continue with the procedure. Each time the response to the prompt is "N", another file name will be presented and the DELETE prompt will be repeated. Each time the response to the prompt is "Y", the indicated file will be deleted and the next file name will be presented along with the DELETE prompt. The command will automatically end when the list of file names has been exhausted.

When the Delete File (or any other) command is executed with the Single Disk Processing in effect, the disk drive number does not need to be specified.

The last command in the category of File Commands is the Modify File Protection command. Important files that are stored on a diskette may be designated as such by the "protected" status.* Protected files cannot be deleted or renamed unless the protected status has been overridden.

When the Modify File Protection command is executed, the disk drive number must be specified. The name of a file must also be specified. When this step has been completed, the following prompt will appear.

PROTECT (Y/N)?

---

*This status should not be confused with the keyword PROTECTED used in Extended BASIC. This is not the same type of protection.

The selection made in response to this prompt will determine the status of a file until it has been changed again with this command.

When a file is copied, the status of the original file will be transferred to the new file as well. For example, if DATA is the name of a protected file on one diskette, and it is copied on another diskette with the name RESULTS, the new file (RESULTS) will be protected as well.

Disk Commands

The Disk Commands are used to perform procedures that affect an entire diskette. The four Disk Commands have the following names.

CATALOG DISK
BACKUP DISK
MODIFY DISK NAME
INITIALIZE NEW DISK

The Catalog Disk command is used to display a list of all the files on a diskette. When this command is executed, the disk drive number must be specified unless the Single Disk Processing mode is in effect.

The next prompt that appears indicates that the list of files can be output to the display thermal printer, RS-232 interface or some other device. Once the device selection has been chosen, the procedure will be ready to begin.

When FCTN-6 (PROC'D) is entered, the list of file names, types of files, file lengths and file protection status will be output. Also, the number of sectors in use and the number of available sectors will be indicated.

The Backup Disk command is used to make a second copy of the contents of a diskette. When the Backup Disk command is executed, a prompt will appear on the display as follows.

SELECTIVE (Y/N)?

This selection will determine whether an entire diskette is to be copied, or only individual files. If the response to this prompt is "Y", each file name will be listed individually, along with its length, type and protection status. The following prompt will then appear on the display.

COPY FILE (Y/N)?

The response to this prompt will determine whether the file is to be copied. After each response, the next file name will be displayed along with the file parameters.

The COPY FILE will be displayed repeatedly until the entire list of file names has been exhausted.

If the reply to the SELECTIVE prompt was "N", the entire contents of the diskette will be duplicated. This procedure is simple if two or more disk drives are in use. In this case, simply specify the disk drive number that contains the master diskette and enter the disk drive number that contains the backup diskette.

Unfortunately, the Backup Disk procedure is rather inconvenient when a single disk drive is in use. In this case the Single Disk Processing mode must be in effect. Each time a file is copied, the master diskette and backup diskettes must be swapped. A series of prompts will be displayed to guide the user through the procedure. However, if several files need to be duplicated, this procedure may become tedious and time consuming.

The third command in the category of Disk Commands is the Modify Disk Name command. This command is used to change the name that was assigned to a diskette. The diskette name is merely a means of identifying the diskette.

When this command is executed, the disk drive number must be specified. The current name of the diskette will be displayed as well as a prompt that indicates that a new diskette name should be entered. When FCTN-6 (PROC'D) is typed, the name of the diskette will be changed.

The Initialize command is the most important Disk Manager command because diskettes cannot be used until they have been initialized. As a result, the Initialize command must be used before any other disk operations can be performed.

When this command is executed, the disk drive number must be specified if the Single Diskette Processing mode is not in effect. The next message will indicate the current name of the diskette, or will indicate that the diskette has not been initialized and does not have a name. At this point, a new diskette name must be specified. The next prompt will appear as follows.

40 TRACKS (Y/N)?

A "Y" response to this prompt will cause 40 tracks of 9 sectors each to be arranged on the diskette. If the response to this prompt was "N", the diskette will be arranged in 35 tracks. Since 40 tracks allow the storage of more data, it is generally advisable to use the 40 track format.

The initializing procedure includes a process that verifies the condition of the diskette. As each sector is examined, the number of the sector will be displayed. If a problem occurs, a message will be displayed on the screen. If such a problem occurs, the initialization should be repeated. If the problem persists, the diskette should not be used.

When the initialization has been completed, the total number of available sectors will be displayed. It should be noted that the initialization procedure eliminates all of the information that may have been stored on a diskette before it had been initialized. Fortunately, a diskette only needs to be initialized once before it is used for the first time.

Disk Tests

There are two types of tests that can be performed for the Disk Memory System: quick tests and comprehensive tests. Quick tests are performed to determine the condition of a diskette. The

comprehensive tests are performed to determine the condition of the entire Disk Memory System, including the Disk Controller and Disk Drive as well as the diskette.

A disk test can be either destructive or non-destructive. A destructive test is a procedure that eliminates all of the information on a diskette. This type of test is usually performed with new diskettes to be sure that they are not damaged.

A non-destructive test does not eliminate any of the information stored on the diskette. As a result, this type of test is generally more useful.

When the Disk Test selection is chosen from the Disk Manager Menu, the following two selections will appear on the display.

1 QUICK TEST
2 COMPREHENSIVE TEST

The Quick test can be chosen at any time, regardless of the condition of the diskette. The first prompt that appears on the display requests the type of test to be performed. If a non-destructive test is desired, enter "N". However, if the diskette has not been initialized, the destructive test must be selected.

If the diskette had not been initialized previously, a prompt will request information concerning the number of tracks on the diskette and the number of the drive containing the diskette. A subsequent prompt will be displayed to determine whether the test should be performed repeatedly or only once. The final prompt will be used to select a device that can be used to record any errors detected during the test.

When a complete set of selections have been made, press FCTN-6 (PROC'D) to begin the test procedure. Press FCTN-4 (CLEAR) if the test needs to be stopped at any point.

A quick test requires only a few minutes to be executed, but a comprehensive test requires about 15 minutes to be completed. Also, a comprehensive test is always destructive. Since this test requires so much time to be completed, it is used only infrequently.

The information required to perform a comprehensive test will be presented with a set of prompts in a similar manner to the quick test.

SAVING AND LOADING PROGRAMS

There are two BASIC commands that are commonly used to save and load programs. These commands are SAVE and OLD. The SAVE command is used when programs are to be stored, and the OLD command is used to recover previously stored programs.

SAVE

When the SAVE command is used to store a program, the command must include a device name as well as a file name. Appropriate device names include DSK1, DSK2 and DSK3. Only the lower number device names may be used if fewer than three disk drives are in use.

The Disk Memory System must be powered on and ready to operate when the SAVE command is executed.

The SAVE command stores the program that currently resides in the computer's memory in a disk file with the specified name. If a file already exists on the specified diskette with the specified file name, the old file will be erased and the new file will be stored in its place.

The following example commands demonstrate the technique used to save programs in disk files.

```
SAVE DSK1. PROGRAM
SAVE DSK2. SORT
SAVE DSK1. FORECAST
```

Before attempting to save a program in a diskette file, be sure that the diskette has been initialized. The initialization procedure can be performed by the Disk Manager Disk Command number 4.

The SAVE command cannot be used unless a program actually exists in the computer's memory. If no program is present, a CAN'T DO THAT error will occur.

OLD

The OLD command is used to recover programs that had previously been saved. Whenever a program is loaded into the computer's memory by means of an OLD command, any program that may have previously been present in the computer's will be erased.

When the OLD command is used to recover a program, the command must include a device name as well as a file name. Appropriate device names include DSK1, DSK2 and DSK3. Only the lower number device names may be used if fewer than three disk drives are in use.

The Disk Memory System must be powered on and ready to operate when the OLD command is executed.

The following example commands demonstrate the techniques used to recover programs from disk files.

```
OLD DSK1. PROGRAM
OLD DSK2. SORT
OLD DSK1. FORECAST
```

Any attempt made to load a program that has not been saved on a diskette will result in DISK ERROR 57.

SAVING DATA

The TI Disk Drives are not restricted to saving and loading programs. They can also be used to save data. Any string or numeric values that need to be saved can be recorded and retrieved with a disk drive.

There are 4 BASIC commands that are used to store and retrieve data. These statements are OPEN, CLOSE, PRINT# and INPUT#.

TYPES OF DATA FILES

A data file is merely a collection of records, and a record is a collection of data items. The records in a data file can be arranged in one of two ways.

A sequential file is a collection of records in which each record can only be accessed in the order in which it was written. In other words, the records are written to the file in a specific order, and they can only be accessed in the same order. The records in a cassette tape data file are an excellent example of a sequential file.

A relative file is a collection of records in which each record can be accessed individually and in any order. Unfortunately, the added convenience of a relative file is counteracted by a reduced efficiency in the use of disk storage space.

Relative files cannot be used unless each record is the same length. However, sequential files can vary in length, as long as an upper limit is not exceeded. Since data items are usually not uniform in length, relative files usually result in a great deal of wasted memory space because the data file contains a large number of records, very few of which are fully occupied.

On the other hand, sequential files can be created so each record is only as long as the data that needs to be stored in that record. This technique results in a much more efficient use of disk memory space.

OPEN

Before data can be output to any external device, an OPEN statement must be executed. The OPEN statement must include the information that is required to establish communication between the computer and the external device.

A filenumber is a parameter that is used to specify a particular channel of communication between the computer and a particular device. It is necessary to use a filenumber because several different peripheral devices may be in use at the same time, or the same device may be used for several functions simultaneously. To avoid confusion, each input or output operation of the computer must be assigned a unique number.

The filenumber must be an integer from 1 to 128, and must be the first parameter in an OPEN statement. Any subsequent PRINT#, INPUT# or CLOSE statement must include the specified filenumber.

An OPEN statement must also include several additional parameters to specify the exact nature of the transfer of data, as well as the format of the data file.

As mentioned earlier, there are primarily two types of data files: relative and sequential. The keyword RELATIVE is used in an OPEN statement to specify a relative file. Similarly, the keyword SEQUENTIAL specifies a sequential file. With all disk data files the keyword INTERNAL should also be specified. This parameter specifies a compact means of storing the data that uses the storage space most effectively.

Sequential files can be opened for either INPUT, OUTPUT, or a special function called APPEND. The keyword INPUT refers to the situation in which data is being retrieved from a file. The keyword OUTPUT is used when data is sent to a new file. The APPEND function is used when it is necessary to add additional records to the end of a sequential file.

An OPEN statement also requires a parameter that specifies the length of each record in the file. Since the nature of sequential files requires variable length records, the keyword VARIABLE should be used with all sequential files.

Relative files can be opened for either INPUT, OUTPUT or a special function called UPDATE. The keyword INPUT once again refers to the situation in which data is being retrieved from a file. The keyword OUTPUT is used when data is to be sent to a new file.

The UPDATE function is used when it is necessary to change the data stored in a record within the file. The structure of relative files requires that each record be the same length. As a result, the keyword FIXED must be included in an OPEN statement for a relative file.

If a record length is not specified, each record in a relative disk file will be 80 characters long. If a smaller record length is desired, the record length can be specified immediately after the FIXED parameter. The following example statements depict the correct format of OPEN statements.

```
OPEN #4: "DSK1. RESULTS", OUTPUT, SEQUENTIAL, VARIABLE,
         INTERNAL
OPEN #7: "DSK2. DATA", OUTPUT, RELATIVE, FIXED 50,
         INTERNAL
OPEN #4: "DSK1. RESULTS", APPEND, SEQUENTIAL, VARIABLE,
         INTERNAL
OPEN #7: "DSK2. DATA", UPDATE, RELATIVE, FIXED 50,
         INTERNAL
OPEN #4: "DSK1. RESULTS", INPUT, SEQUENTIAL, VARIABLE,
         INTERNAL
OPEN #7: "DSK1. DATA", INPUT, RELATIVE, FIXED 50,
         INTERNAL
```

CLOSE

A CLOSE statement is used to eliminate a channel that had been previously established for an input or output operation. A CLOSE statement does not require any parameters other than the filenumber.

A CLOSE statement is not always required since each input or output channel will be automatically closed when the program ends. However, it is a good programming practice to close each channel that has been opened in a program.

A typical CLOSE statement would have the following structure.

```
CLOSE #1
```

A CLOSE statement can include the keyword DELETE. This statement causes the data file that is being closed to be deleted as well. Be sure to include a colon after the filenumber, as depicted below.

```
CLOSE #1: DELETE
```

DELETE

A DELETE statement is used to delete files from a diskette. The only parameters required for a DELETE statement are an appropriate device name and file name. As a result, a typical DELETE statement would appear as follows.

```
DELETE "DSK1. OUTPUT"
```

A disk error will occur if a DELETE statement specifies a nonexistant device.

PRINT#

A PRINT# statement is used to output data to a diskette data file. The format for this statement, is basically the same as the PRINT

statement, except for the filenumber. Each PRINT# statement must include a filenumber, followed by a colon. The values that follow the colon will be output to the specified file.

For example, the following PRINT# statement would cause the values of the variables A$, B$ and C$ to be output to a data file

```
PRINT# 1: A$, B$, C$
```

When data is output to a file, the data items in the PRINT# statement should be separated by commas.

The data is output in sections called records. A record is merely a section of the diskette that is used to store one or more data items. Each PRINT# statement outputs exactly one record.

For diskette files, each record will be 80 characters long unless a smaller number was specified in the corresponding OPEN statement. As a result, the data being sent to the data file cannot exceed a total of 80 characters.

The length of a string value is the total number of characters in that string. Letters, numbers, spaces and punctuation marks all contribute to the length of a string value. Each numeric value has the equivalent length of an 8 character string value.

One space of the data record must be used to separate the individual values. These spaces must also be considered when record lengths are determined. As a result, the total number of characters in the record can be calculated as follows.

```
    Total number of variables
    Total number of characters in string values
+   Eight characters for each numeric value
    ----------------------------------------
    Total record length.
```

Writing to Sequential Files

Sequential files contain a set of records that are recorded one after the other, in a specific order. As a result, the records do not

need to be of uniform length. The keyword VARIABLE should be included in the OPEN statement that corresponds to the particular file.

Since the records of a sequential file are automatically stored with the correct length, the record length is of no concern to the programmer as long as the maximum length of 80 characters is not exceeded. A typical PRINT# statement for a sequential file would appear as follows.

```
PRINT #1: A, B, C$
```

Writing to Relative Files

Since the records of a relative file can be accessed in any order, a PRINT# statement for this type of file must specify the number of the record that is to receive the data. The REC statement in a PRINT# statement performs this function. A REC statement must be included after the filenumber, but before the data items in a PRINT# statement. The following example statement portrays the typical structure of a PRINT# statement for a relative file.

```
PRINT #1, REC 4: A$, B$, C$
```

The example statement would cause the values of the variables A$, B$, and C$ to be output to record number 4 of the file opened for channel number 1. Keep in mind that the first record in a relative file is record number zero.

The nature of relative files requires that the records in a file be of uniform length. As a result, the most efficient use of diskette storage area requires the minimum record length. However, the records must all be long enough to accomodate the amount of data that appears in the longest record of the file. The FIXED statement that appears in every OPEN statement for a relative file may be followed by any value less than or equal to 80. This value determines the length of every record in the entire file.

INPUT#

An INPUT# statement is used to retrieve data from a diskette data file. The format for this statement is basically the same as the INPUT statement, except for the filenumber. Each INPUT# must include a filenumber followed by a colon. The variables specified in the INPUT# statement are assigned values that have previously been stored in a data file.

For example, the following INPUT# statement would cause values to be input for the variables X$, Y$, and Z$.

```
INPUT #1: X$, Y$, Z$
```

The variables specified in an INPUT# statement must be separated by commas.

Data is stored in a data file as a collection of records. Each record in the file has a specific format. The type of data in each record is determined by the structure of the PRINT# statement that was used to output the data. For example, if a PRINT# statement outputs two string values followed by two numeric values, these values are recorded in a specific order in a specific record.

As a result, the data in the field can only be recovered by an INPUT# statement that has the same structure. This principle is demonstrated in Illustration 7-5.

Illustration 7-5. Corresponding PRINT#/INPUT# Formats

output statement	corresponding input statement
PRINT #1: A$	INPUT #4: X$
PRINT #27: A$, B$, C, D	INPUT #3: L$, M$, N, O
PRINT #1: A, B, C, D	INPUT #1: A, B, C, D
PRINT #4: A, B$, C	INPUT #6: X, X$, Y

There are several important considerations in the use of PRINT# and INPUT# statements. The most important detail is the fact that each corresponding pair of PRINT# and INPUT# statements must have the same number of variables. String variables must correspond to string values and numeric variables must correspond to numeric values.

The actual names of the variables do not have to be exactly the same in a corresponding pair of statements. The only requirement is that the types of variables must be consistent. Similarly, the filenumbers in corresponding statements do not have to be the same, although this is often the case.

Reading from Sequential Files

INPUT# statements are used to recover data from a diskette file in sequential order. The data is assigned to the variables that are listed in the INPUT# statement. The data must be input in exactly the same order in which it was recorded.

An INPUT# statement must include exactly the same number and type of variables as the corresponding PRINT# statement.

Reading from Relative Files

The data stored in relative files can be accessed in any order. As a result, a record number can be specified when an INPUT# statement is used with relative files. The keyword REC can be included in the INPUT# statement immediately after the filenumber.

An INPUT# statement can be used with relative files without a record number. However, if no record was specified, the data will be taken from the record immediately after the one that had been accessed previously. As a result, data can be recovered from a relative file in a sequential style if no record numbers are specified. The following two example statements demonstrate the format of typical INPUT# statements.

```
INPUT #1, REC 2: A, B, C
INPUT #1: A, B, C
```

The EOF Function

The EOF function returns a numeric value that indicates if the end of a file has been reached. This function requires an argument that corresponds to the filenumber of a diskette data file. A typical program statement that uses the EOF function would appear as follows.

```
PRINT EOF (1)
```

The EOF function is used most often when sequential files are used as the input to a program. If the end of the file has not been reached, the function will return a zero value. If a problem arises that does not allow the input process to continue, the function will return a value of -1. If the end of a file has been reached by the normal exhaustion of data, the function will return a value of 1.

Recall that an IF, THEN statement considers any non-zero value to be a true condition. As a result, a statement with the following format can be used to branch the execution of a program when the end of a file has been reached.

```
IF EOF (1) THEN 450
```

The RESTORE Statement

A RESTORE statement can be used to alter the order in which the records in a sequential file are accessed. Normally, the records in a sequential file are accessed from the first record in the file to the last. However, a RESTORE statement causes the first record in the file to be accessed again. In other words, a RESTORE statement has the same effect as closing a file and reopening it again. A RESTORE statement must always contain the filenumber that corresponds to a diskette data file. The following statement demonstrates the format of a typical RESTORE statement for a sequential file.

```
RESTORE #1
```

A RESTORE statement can be used with relative files in a similar manner. However, this statement is only of limited use since the records in a relative file can always be accessed in any order. When used with a relative file, a RESTORE statement may include a REC statement that specifies a particular record in a file. As a result, the next record accessed in the file will correspond to the record number specified in the RESTORE statement. The following example statement demonstrates a typical RESTORE statement for a relative file.

```
RESTORE #1, 27
```

Sequential Data File Example

An example program is included here in order to present the techniques used to create and use a data file. This example includes a sequential file arranged in a possible format for a parts order form.

```
10 OPEN #1: "DSK1. ORDER", OUTPUT, SEQUENTIAL,
   INTERNAL, VARIABLE
20 CALL CLEAR
30 INPUT "DESCRIPTION?":D$
40 INPUT "PART NUMBER?":P
50 INPUT "QUANTITY?":Q
60 PRINT #1:D$, P, Q
70 INPUT "ENTER S TO STOP": S$
80 IF S$ <> "S" THEN 20
90 CLOSE #1
100 INPUT "ENTER R TO REVIEW": R$
110 IF R$ <> "R" THEN 200
120 CALL CLEAR
130 PRINT "PARTS ORDER":::
140 OPEN #1: "DSK1. ORDER", INPUT, SEQUENTIAL,
    INTERNAL, VARIABLE
150 INPUT #1: A$, B, C
160 PRINT A$, B; C
170 IF EOF (1) THEN 190
180 GOTO 150
190 CLOSE #1
200 END
```

The program has two major parts; one for writing data to a file and one for recovering the data. The first part begins with an OPEN statement that assigns filenumber 1 to the data file DSK1. ORDER. The prefix DSK1 indicates that the files should be created on the diskette in disk drive number 1. The OPEN statement also specifies that the file will be written in a sequential fashion with variable length records.

The program proceeds with three INPUT statements that present prompt messages on the display and accept data entered at the keyboard. A PRINT# statement is used to write the three data items in a single record of the file. An additional INPUT statement is included to determine the end of the list of data. In order to stop the data entry routine, the letter S must be entered in response to the prompt. If any entry other than S is made the program will request additional data.

The second part of the program begins with an INPUT message that asks if the contents of the file need to be reviewed. If the response to this prompt is anything other than the letter R, the program will branch to the end of the program. If R was entered, the data file DSK1. ORDER will be opened once again. However, the file will be opened for input rather than output. Notice that the file's other parameters will match those identified in the initial file access.

Before the contents of the file are read, the display will be cleared and the message "PARTS ORDER" displayed. Once the file has been opened, an INPUT# statement will be repeated until the end of the file has been detected. Notice that the number of variables and type of variables in the INPUT statement corresponds exactly to the PRINT# statement in the first part of the program. When the data in the file has been exhausted, the file will be closed and the program will end.

The records of a sequential file cannot be altered once the file has been created. However, additional records can be added to the end of the file if the APPEND operation was used. When an OPEN statement is used to reopen a sequential file, the para-

meters of the file must once again be exactly the same. The following example statement could be used to reopen the file DSK1. ORDER.

```
OPEN #1: "DSK1.ORDER", APPEND, SEQUENTIAL, INTERNAL,
         VARIABLE
```

Relative Data File Example

The following example program demonstrates a technique that can be used to create and maintain a relative file.

```
10 OPEN #1: "DSK1. DATA", UPDATE, RELATIVE, FIXED,
   INTERNAL
20 CALL CLEAR
30 X = X + 1
40 INPUT "NAME:":N$
50 INPUT "ADDRESS:":A$
60 INPUT "CITY:":C$
70 INPUT "STATE:":S$
80 PRINT #1, REC 0:X
90 PRINT #1, REC X:N$, A$, C$, S$
100 INPUT "ENTER S TO STOP": STOP$
110 IF STOP$ = "S" THEN 130
120 GOTO 20
130 CLOSE #1
140 INPUT "ENTER L TO LIST NAMES":L$
150 IF L$ <> "L" THEN 240
160 CALL CLEAR
170 OPEN #1: "DSK1. DATA", INPUT, RELATIVE, FIXED,
    INTERNAL
180 INPUT #1, REC O:X
190 FOR N = 1 TO X
200 INPUT #1: A$, B$, C$, D$
210 PRINT A$: B$: C$: D$:::
220 NEXT N
230 CLOSE #1
240 END
```

The preceeding example program is divided into two parts. The first part creates the data file and allows data to be entered into the file. The second part allows the data to be recalled and displayed on the screen.

The program begins by opening filenumber 1 for the disk data file DSK1.DATA. Data can be written to the file as well as read because the file is a relative file opened for the UPDATE operation.

The variable X in this program is used to keep track of the current record number in the file. As a result, the value of X must be increased each time a new record is added to the file. Record number zero in the data file is used to store the number of records that are contained in the file.

The input routine in the first part of the program provides prompts and accepts four values to be stored in each record. When all four values have been entered, the current record number will be stored in record number zero, and the data will be stored in the next available record. An additional INPUT statement was included to allow the user to end the input routine.

When the input routine has been completed, the data can be recovered from the file by entering the letter L in response to the "LIST NAMES" prompt.

In order to recover the data, the file must be opened once again. The same file parameters must be specified each time a file is opened. Record number zero must be read first in order to determine the total number of records in the file. When this value has been determined, a FOR, NEXT loop will be established to read the data from the file. An INPUT# statement within the loop will be used to read the four values from each record of the file.

The loop will be repeated until each record in the file has been read and displayed. When this procedure has been completed, the file will be closed and the program will end.

CALL FILES

Each file that is opened for input or output to the computer will consume a portion of the computer's memory. As a result, TI BASIC imposes a limit of 3 files that can be simultaneously open.

If it becomes necessary to use more than 3 files, the CALL FILES statement can be used to extend the limit to any number less than 10. A CALL FILES statement must be executed in the immediate mode, followed by the NEW command. A CALL FILES statement must not be used as a statement in a program.

The following two statements demonstrate the technique used to allow 5 files to be simultaneously open in a program.

```
CALL FILES (5)
NEW
```

Extended BASIC Features

Extended BASIC allows the use of a LINPUT# statement to input an entire record of a file. Unfortunately, this statement can only be used with files that had been recorded with the DISPLAY format rather than INTERNAL. With relative files, a LINPUT# statement can include a REC statement in order to specify a particular record within a file. The following two example statements depict the two formats of a LINPUT# statement.

```
LINPUT #1, REC 27:A$
LINPUT #1:A$
```

A LINPUT# statement can only include a single string variable name since the entire record will be returned as one value.

Extended BASIC also allows the use of the REC function. This function returns the number that corresponds to the record that will be accessed next in a relative file. In other words, the REC function returns a value that is one greater than the number of the current record.

The REC function requires an argument that corresponds to the filenumber of a currently opened data file.

CHAPTER 8. THE TI-99/4A PRINTER

The TI-99/4 printer (model PHP 2500) allows its user to output programs and data using virtually any standard computer form. This printer includes a number of useful features--including graphics capabilities and a variety of type styles that allow data to be output in any one of several different formats.

The computer can send data to the printer in one of two modes. These two modes are commonly called **serial** and **parallel.** Regardless of the mode, the printer must be used with an RS232 interface module. The TI-99/4 printer is most commonly used in the serial mode.

Early models of the RS232 interface module do not have a parallel output. As a result, the printer must be used in the serial mode with this style of interface module.

New models of the RS232 interface have both serial and parallel output. This type of interface is installed in the Peripheral Expansion box.

The cable supplied with the printer allows it to be used in the serial mode. The printer can only be used in the parallel mode if a special cable is purchased.

Installation

Carefully follow the installation instructions that appear in the printer manual. These instructions provide a complete guide to the installation of the printer.

When the installation is complete, be sure to test the printer by holding down the LF button and turning the printer's power on. If the results of the test are satisfactory, and the printer is properly installed, the printer is ready to be operated.

The remainder of this chapter is presented in three sections. The first section deals with the BASIC commands that can be used to list programs and output data to the printer. The second section is dedicated to the special printer commands that control the special functions of the printer. The final section of this chapter provides an explanation of the techniques used to generate graphics with the printer.

Listing Programs

The LIST command can be used to output a copy of the program that is currently stored in the computer's memory.

Since the printer is connected to the RS232 interface, the LIST command requires the device name "RS232" to cause the output to be sent to the printer.

The following example depicts the command that is used to output the entire program that is stored in the computer's memory.

```
>LIST "RS232"
```

The LIST command can also be used with line numbers to specify portions of the program to be output. For example, the following command would cause program lines 100 through 1000 to be output to the printer.

```
LIST "RS232": 100-1000
```

Outputting Data

A PRINT# statement is most commonly used to output data to the printer. However, an I/O channel must be opened for the printer before any data can be output.

An OPEN statement is required to open an I/O channel. An OPEN statement for the printer requires a filenumber as well as the device name "RS232". The filenumber can be any integer from 1 to 255.

The following statement is a typical OPEN statement that can be used to establish an I/O channel for the printer.

```
OPEN #7: "RS232"
```

Any subsequent PRINT# statements in a program that specify filenumber 7 will cause the output to be sent to the printer.

The following example program demonstrates the use of OPEN and PRINT# statements to output data to the printer.

```
10 OPEN #7: "RS232"
20 FOR J=1 TO 20
30 PRINT #7: J, J^2
40 NEXT J
50 CLOSE #7
60 END
```

The preceding example program causes the values from 1 to 20 to be output, along with the squares of these values.

PRINT# statements are used in a similar manner to PRINT statements. The only difference between the two types of statements lies in the use of the filenumber.

When the data values in a PRINT# statement are separated by commas, the output will be arranged in columns. Six columns of data can be output on a standard 8½" paper width.

When the data values in a PRINT# statement are separated by semicolons, no additional spaces are inserted in the output. String values are output adjacent to each other, but numeric values are always preceded and followed by one blank space.

A CLOSE statement is used to prevent access to an I/O channel that was previously opened with an OPEN statement.

A CLOSE statement requires the filenumber of the I/O channel that is no longer needed. CLOSE statements are not always necessary since all I/O channels are automatically closed when the program ends. However, it is a good programming practice to include CLOSE statements that correspond to each OPEN statement in a program.

Printer Commands

The TI-99/A printer's output can be controlled using a variety of different printer commands.

For example, by outputting certain characters to the printer, the programmer can vary the typestyle used to output data. Printer control characters also allow the programmer to set tab stops, vary the character set, control line spacing, control the form length, and control the line length. Each of these printer control commands will be discussed in the next few pages.

These printer control characters can be generated with the CHR$ function. For example, character number 13 causes the printer to return to the beginning of the next line. This is known as a carriage return. The following example contains a typical PRINT # statement that generates a carriage return.

```
PRINT #2: CHR$(13)
```

Many of the special functions of the TI-99/4 printer require more than one character to be specified. When a special function requires more than one printer control character, the CHR$ statements should be separated by semicolons in a single PRINT# statement.

For example, the following statement would be used to output four special characters to the printer.

```
PRINT #3:CHR$(27);CHR$(76);CHR$(80);CHR$(0)
```

In the preceding example, the characters CHR$(76) and CHR$(80) would be output to the printer. The ASCII values 76 and 80 correspond to the letters L and P respectively. As a result, it was not actually necessary to use the CHR$ function to generate these two characters.

The following three statements have the same effect as the previous example statement.

```
PRINT #3 :CHR$(27); "L"; CHR$(80) ; CHR$(0)
PRINT #3 :CHR$(27); CHR$(76); "P"; CHR$(0)
PRINT #3 :CHR$(27); "LP"; CHR$(0)
```

Print Styles

The TI-99/4 printer can output characters in any one of 12 distinct styles. These styles can be controlled by a program through the use of special printer control characters.

The two fundamental type styles are called standard and condensed. The standard print style is automatically used when the printer is first powered on. This style outputs 10 characters per inch for a total of 80 characters per line.

The condensed print style outputs approximately 17 characters per inch for a total of 132 characters per line. The standard and condensed print styles cannot be used on the same line of output.

Either of these fundamental styles can be enlarged to twice their normal width. As expected, only half as many double width characters can be output on each line. Both enlarged and normal width characters can be output on the same line.

The TI-99/4 printer can also be used to output bold characters. There are two techniques that are used to generate these print styles: double printing and emphasized print.

When double printing is used, the printer types each character twice in order to make a darker impression on the paper. The double printing mode can be used in conjunction with any of the print styles. Unfortunately, the printer must reduce its rate of output when the double print mode is used.

Emphasized print is similar in principle to double printing, but emphasized print does not reduce the speed of the printer. In the standard output mode (10 characters per inch), emphasized characters are the same size as non-emphasized characters. However, in the condensed output mode (17 characters per inch), emphasized characters are larger. Emphasized condensed characters are output in the standard character size (10 characters per inch).

As a result, the TI-99/4 printer can output characters in any of four sizes. Table 8-1 summarizes the sizes of these four print styles.

Table 8-1. Print Sizes

Style	Characters/inch
Condensed	17
Standard or Emphasized condensed	10
Enlarged condensed	8.5
Enlarged Standard or Enlarged Emphasized Condensed	5

Generally, when a print style is selected, it will be in effect for the entire line of output. However, enlarged characters can be used at any time without disrupting the entire line of output.

Table 8-2 includes a complete list of the commands used to activate and deactivate the print styles.

Table 8-2. Print Style Commands

Mode	Activate	Deactivate
Standard	CHR$(18)	
Condensed	CHR$(15)	CHR$(18)
Enlarged	CHR$(14)	CHR$(20)
Double	CHR$(27);"G"	CHR$(27);"H"
Emphasized	CHR$(27);"E"	CHR$(27);"F"

The following example program demonstrates the technique used to output characters in four different sizes.

```
100 OPEN #1:"RS232"
200 OPEN #1:CHR$(15);"CONDENSED"
300 PRINT#1:CHR$(27);"E";"EMPHASIZED"
400 PRINT #1:CHR$(27);"F";
500 PRINT #1:CHR$(14);"ENLARGED"
600 PRINT #1:CHR$(14);CHRS$(27);"E"
700 PRINT #1:"ENLARGED EMPHASIZED"
800 CLOSE #1
900 END
```

Character Sets

The TI-99/4 printer can use any one of 8 character sets. All 8 character sets are similar to each other, but each one contains several unique characters. The character sets are commonly referenced by the country for which the character set is most appropriate.

The eight character sets are designated by the values 0 through 7 as listed in Table 8-3.

The character set for England contains only one character that differs from the U.S.A. character set. In the English character set, the £ symbol is included instead of #. As a result, when the English character set is in use, the £ symbol is printed each time the # character is sent to the printer.

In order to select a character set other than character set number zero, a three character command must be executed. The first two characters must be CHR$(27) and CHR$(82) respectively. The third character must correspond to the desired character set. For example, the following statement could be used to select the character set for Italy.

```
PRINT #1: CHR$(27);CHR$(82);CHR$(6)
```

Table 8-3. Character Set Values

Country	Value
U.S.A.	0
France	1
Germany	2
England	3
Denmark	4
Sweden	5
Italy	6
Spain	7

Tabulation

The TI-99/4 printer allows tab stop locations to be set for the lines of output as well as for columns. The horizontal tab function causes the printer to proceed to a specified column location. The vertical tab function causes the printer to proceed to a specified line on the page.

The horizontal tab function can be used to set up to 12 tab stop locations. In order to set the tabs, CHR$(27) and CHR$(68) must be sent to the printer. Additional characters must be sent to the printer to specify the tab locations. The last character must be CHR$(0). Be certain that the tab locations are listed in ascending order. The following characters can be used to set the tab locations at column numbers 30, 60 and 70.

```
CHR$(27);CHR$(68);CHR$(30);CHR$(60);CHR$(70);CHR$(0)
```

CHR$(9) is the command that causes the printer to proceed to the next tab stop location. If the CHR$(9) command was executed when the printer had passed the last tab stop on a line, the printer would ignore this command.

The tab function will have no effect when enlarged characters are being output.

Vertical tabulation is similar in principle to horizontal tabulation. Vertical tabs correspond to line numbers rather than column numbers. The first two characters of a vertical tab statement must be CHR$(27) and CHR$(66). The last character must be CHR$(0). As many as 8 vertical tab locations can be used at one time.

The following characters can be used to set tab locations at lines 30, 40 and 60. Once again, be certain that the line numbers are specified in ascending order.

```
CHR$(27);CHR$(66);CHR$(30); CHR$(40);CHR$(60);CHR$(0)
```

The CHR$(11) code is used to cause the printer to proceed to the next predetermined line number. The lines on a page range from 1 to 66.

In order for the vertical tab function to operate properly, the paper must be set with the top of the page in the correct location. This setting must be made before the printer is powered on.

The following example demonstrates the use of the horizontal and vertical tab functions. Use the FF key to advance the printer to the top of the next page before beginning this program.

```
100 OPEN #1: "RS232"
110 FOR J=0 TO 8
120 READ X
130 PRINT #1:CHR$(X);
```

```
140 NEXT J
150 DATA 27, 68, 35, 0
160 DATA 27, 66, 33, 65, 0
170 PRINT#1:CHR$(9);"TOP CENTER"
180 PRINT#1:CHR$(11);"MIDDLE LEFT"
190 PRINT #1:CHR$(11); CHR$(9);"BOTTOM CENTER"
200 CLOSE #1
```

The FOR/NEXT loop at lines 120 to 140 is used to send 9 special characters to the printer. This technique is usually more convenient to use than individual CHR$ functions.

The DATA statement at line 150 provides the character codes that would be used to set a tab stop at column number 35. The DATA statement at line 160 provides the character codes that will be used to set tab stops at line numbers 33 and 65.

When the program is executed, the PRINT statement at line 170 will cause the message "TOP CENTER" to be output at the center of the first line. The PRINT statement at line 180 will cause the printer paper to advance to the 33rd line of the page. The message "MIDDLE LEFT" is output at this location. Line 190 will cause the printer paper to advance to the last line of the page. The message "BOTTOM CENTER" will be printed in the center of the last line.

Line Spacing

The line spacing of the printer can be set to either 8 or 6 lines per inch. When the printer is powered on, the spacing is set to 1/6 inch. The following command can be used to cause narrow line spacing.

```
CHR$(27);CHR$(48)
```

In order to return to wide spacing, execute the following command.

```
CHR$(27);CHR$(50)
```

Form Length

The printer is generally used with paper that is eleven inches long. However, if a different size paper is used, the printer can be informed of the new paper length. This allows the form feed and vertical tabulation functions to operate properly regardless of the length of the paper.

The following command can be used to specify a form length of up to 127 lines. The number of lines is represented by X.

```
CHR$(27);CHR$(67);CHR$(X)
```

Line Length

The maximum line length can be specified with a three character code. The first two characters must be CHR$(27) and CHR$(81). The last character must correspond to the length of the line. For example, the following command can be used to set a maximum line length of 40 characters.

```
CHR$(27);CHR$(81);CHR$(40)
```

Bottom Margin

Character codes 27 and 78 can be used to set a margin at the bottom of each page. The third character code in the Bottom Margin command specifies the number of blank lines that are to be inserted at the bottom of each page.

The following command can be used to specify a bottom margin of 4 lines.

```
CHR$(27);CHR$(78);CHR$(4)
```

Carriage Return

The Carriage Return command (CHR$(13)), causes the printer to return to the first column of the current line. This feature allows more than one character to be printed at the same location. For example, consider the following program.

```
100 OPEN #1: "RS232"
110 PRINT #1: "ONE";
120 PRINT #1: CHR$(13);
130 PRINT #1: "TWO"
140 PRINT #1
```

The PRINT statement at line 110 causes the word "ONE" to be output. The Carriage Return at line 120 causes the printer to return to the beginning of the same line. The word "TWO" is then output directly on top of the word "ONE".

Line Feed

The Line Feed function (CHR$(10)) causes the printer to advance the paper one line and return the carriage to the first column of the new line.

Form Feed

The Form Feed function (CHR$(12)) causes the printer to advance the paper to the first line of the next page. Any subsequent output will begin at the upper left corner of the new page.

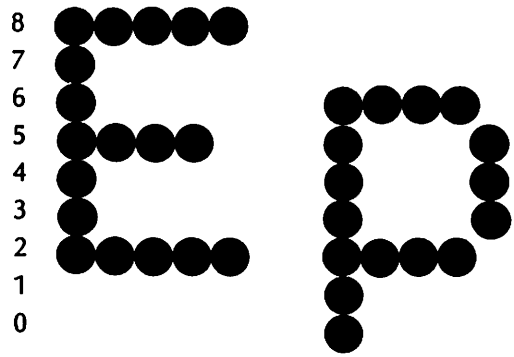
Bell

The CHR$(7) command causes the printer's buzzer to sound.

GRAPHICS

The output of the printer consists of an array of dots. These dots are arranged in such a way as to form a specific character. For example, Illustration 8-1 depicts the dot formations of the characters E and p.

Illustration 8-1. Character Dot Patterns



A total of 9 print elements are used in the formation of the character sets. In the graphics mode, seven* of these dot elements, 1 through 7, can be controlled independently. As a result, any number of different characters or graphic displays can be generated.

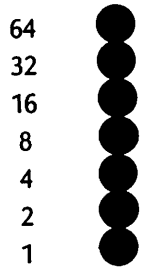
Graphics can be generated in one of two modes: single density or dual density. Single density graphics allows 480 dots to be printed across the width of the page. The dual density graphics mode allows twice as many dots to appear in each row.

Each column of dots is specified by a numeric value from 0 to 127. This value specifies the combination of dots that are to appear in the output. Each print element is specified by the values indicated in Illustration 8-2.

---

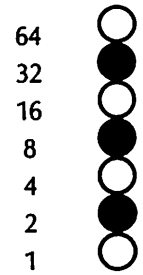
*Eight dot elements may be used if the printer is modified.

Illustration 8-2. Print Element Values



Any combination of the print elements can be specified by the sum of the values of the dots to be output. For example, the pattern in Illustration 8-3 can be specified by the value 42.

Illustration 8-3. Determining Pattern Values



2 + 8 + 32 = 42

The single density graphics mode is activated by the special characters CHR$(27) and CHR$(75). The double density graphics mode is activated by CHR$(27) and CHR$(76).

When either graphics mode is activated, two additional characters must be specified. These characters are used to specify the number of graphics data items that will be supplied.

Because the CHR$ function is limited to 128 values, and the number of data items may be as large as 960, two values are required to specify the number of data items. As a result, the first value is used as an actual number of data items, and the second value indicates additional multiples of 128. The following expression demonstrates the method used to specify the number of data items.

total = *value 1* + *value 2* * 128

Illustration 8-4 provides several examples of the use of this technique.

Illustration 8-4. Specifying the Number of Data Items.

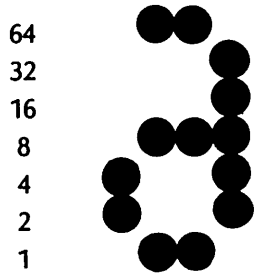
Characters	Value	Expression
CHR$(10); CHR$(0)	10	10 + (0 * 128)
CHR$(100); CHR$(0)	100	100 + (0 * 128)
CHR$(72); CHR$(1)	200	72 + (1 * 128)
CHR$(44); CHR$(2)	300	44 + (2 * 128)

The characters that specify the amount of data must be included directly after those characters that specify the graphics mode.

The following example program demonstrates the technique used to create a graphics characters and insert the character in a section of text.

The first step is to determine the data that defines the character. Illustration 8-5 contains the data required to define a commonly used mathematic symbol.

Illustration 8-5. Data for a Graphics Character



6, 73, 73, 62

Once the data has been determined, that data can be used in a program.

```
100 OPEN #1: "RS232"
110 PRINT #1: CHR$(27); CHR$(75);
120 PRINT#1:CHR$(4);CHR$(0);
130 FOR J= 0 TO 3
140 READ X
150 PRINT #1: CHR$(X);
160 NEXT J
170 CLOSE #1
180 DATA 6, 73, 73, 62
```

The preceding example program can be used to generate a single graphics character. The FOR/NEXT loop specified in lines 130 through 160 is used to send the graphics data to the printer.

Lines 110 and 120 use the CHR$ function to enter the single density graphics mode and specify the number of data items. Since the CHR$ function is used 8 times in this program, the program can be simplified by containing all 8 CHR$ functions in a single FOR/NEXT loop.

If this technique is used, the entire section of the program that generates the special character can be placed in a simple subroutine. This technique allows graphics to be easily mixed with text. The following example demostrates this concept.

```
                100 OPEN #1: "RS232"
                110 PRINT #1: "f (x) =";
                120 GOSUB 1000
                130 PRINT #1: "F/";
                140 GOSUB 1000
                150 PRINT #1: "x"
                160 CLOSE #1
                170 END
                1000 FOR J = 0 TO 7
                1010 READ A
Subroutine      1020 PRINT #1: CHR$ (A);
                1030 NEXT J
                1040 RESTORE
                1050 RETURN
                1060 DATA 27, 75, 4, 0
                1070 DATA 6, 73, 73, 62
```

The preceding example program uses a subroutine to print a graphics character. Each time the subroutine is called, the graphics character will be displayed. The output of the example program should appear as follows.

f (x)= ∂F/∂x

In the double density graphics mode, the printer functions in a similar manner. However, twice as much data is required to define a double density graphics character. This additional effort is compensated by higher quality graphics.

Appendix A. TI BASIC Error & Warning Messages

The TI-99/4A computer can only recognize instructions that are presented in a valid format and do not contain any contradictory or ambiguous information. If a statement does not comply with these restrictions, an error or warning condition will occur.

There are four general categories of errors that can be classified by the situation in which they occur. Many errors occur as soon as the command or statement has been entered. Others occur during the brief period when a program is scanned (prior to execution) for organization errors. The third category of errors occur during the actual execution of a program. The final type of errors occur during the input or output processes of the computer. Each of the error messages will be presented in this appendix according to its category.

STATEMENT ENTRY ERRORS

Statement entry error messages will be displayed immediately after an incorrect statement has been entered. These types of error messages apply only to the most recently entered statement.

Explanations of the statement entry error messages are given in the following sections.

Bad Line Number

This error message will occur when an illegal line number has been specified. Line numbers must be within the range from 1 to 32767.

Bad Name

This error is the result of a variable name in excess of 15 characters.

Can't Continue

This error will result from an incorrect use of the CON command. This command can only be used if a breakpoint in the

execution of a program has occurred. The CON command cannot be used if a program has been halted because of an error, or if a program has been edited after having been halted.

Can't Do That

This error will occur when a statement has been used in the command mode or when a command has been used in the program mode. For example, an EDIT statement cannot be used in a program, and a DATA statement cannot be entered as a command. This error also will occur when a LIST, RUN, or SAVE command was issued when a program was not present in the computer's memory.

Incorrect Statement

This error will occur when a statement is entered with an incorrect format. This error usually results from incorrect punctuation marks.

Line Too Long

This error will occur when a program line contains too many characters.

Memory Full

A Memory Full error will result when an insufficient amount of the computer's memory is available to perform the specified command or accommodate the specified program line.

SCAN ERRORS

Scan error messages will be displayed immediately after the RUN command has been issued. These error conditions are generally the result of incorrect program organization. Scan error messages will be displayed along with the line number of the statement causing the error.

The following sections provide explanations of the scan error messages.

Bad Value

This error is the result of an incorrectly dimensioned array.

Can't Do That

This type of error message will be displayed when a program contains more than one OPTION BASE statement, or when an OPTION BASE statement is preceded by a DIM statement.

For-Next Error

The FOR-NEXT error results from the incorrect use of FOR-NEXT loops. Each FOR statement in a program must have a corresponding NEXT statement.

Incorrect Statement

This error will occur when a statement is used with an incorrect format. Many incorrect statements are not detected upon entry. As a result, these statements will cause errors during the scan process.

Memory Full

A Memory Full error will occur when an insufficient amount of memory is available for variables, arrays, or function definitions.

Name Conflict

This error will occur when a variable, function, or array name duplicates the name of a function, array, or variable. In other words, a variable name cannot be simultaneously used as an array name or a function name, etc.

EXECUTION ERRORS

This category of errors results from error conditions that occur during the actual execution of a program. Execution errors cause the program to be halted at the point at which the error occurred. A line number will always be displayed along with the description of an execution error condition.

The following sections provide explanations of the execution errors.

Bad Argument

This error occurs when a function has been executed with an inappropriate value as an argument.

Bad Line Number

This error will occur when a statement specifies a line number that does not exist in a program. Bad Line Number errors generally occur in the following types of statements: GOTO, GOSUB, IF-THEN, etc.

Bad Name

A Bad Name error will occur when a CALL statement contains an invalid subprogram name.

Bad Subscript

This error will occur when the subscript of an array is inappropriate. This error generally results from a subscript not being an integer, or lying outside of the allowable range of values.

Bad Value

This error message indicates that a statement contains one or more inappropriate values. Since the parameters in many statements are restricted to specific types of values, parameters outside of the allowable range will cause a Bad Value error.

Can't Do That

This type of error will occur when a RETURN statement is encountered without a corresponding GOSUB. Also, a NEXT statement without a corresponding FOR statement can cause this error. A misused BREAK statement can also cause this error.

Data Error

A Data Error results from a problem with a DATA, READ, or RESTORE statement. A program must be organized in such a way that each variable in each READ statement will correspond to a valid data value.

Incorrect Statement

Some statements with an incorrect format are not detected upon entry or upon scanning. As a result, these errors will be detected during program execution.

Memory Full

A Memory Full error occurring during the execution of a program may have been caused in several ways. This error may result from an actual exhaustion of available memory, but it may also have been caused by a programming error. An excessive number of subroutines or function calls may cause a Memory Full error. Also, a subroutine or function that references itself will cause a Memory Full error.

Number Too Big

A Number Too Big warning message will be displayed if a number in scientific notation requires an exponent greater than 127. This warning condition will not terminate program execution.

String-Number Mismatch

This type of error will occur when a string value is used instead of a numeric value, or vice versa.

INPUT/OUTPUT ERRORS

The final category of errors occur during the input or output procedures of the computer. These messages may or may not display a line number indicating the error's location.

Check Program In Memory

This message will be displayed when a problem occurred during the loading procedure. When the OLD command is being used and a problem occurs, the program that previously existed in the computer's memory may or may not be eliminated. However, the OLD command is generally used under the assumption that the current program in the computer's memory will be eliminated in order to accommodate the incoming program. As a result, this message is usually insignificant.

File Error

A File Error occurs when an inappropriate statement has been executed for a data file. For example, an OPEN statement for a file that is already open will cause an error. Similarly, a CLOSE statement for a file that has not been opened will also cause an error.

Input Error

When data is being entered from the keyboard, an Input Error merely causes a warning message. However, an Input Error caused by a data file operation causes program execution to be terminated. INPUT requires a data entry for each variable included in the statement. Also, the type of each data entry must match the corresponding variable in the INPUT statement. If either of these conditions are violated, an Input Error will occur.

I/O Error

This type of error can be caused by a wide range of input or output operations. This type of error message is unique in the fact that it is accompanied by a two digit code that describes the exact nature of the error condition. The first digit of the code specifies the type of statement causing the error condition.

first digit	statement
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

The second digit of the code describes the specific nature of the problem.

second digit	description
0	Device not present
1	Write Protection in effect
2	Improper file specification
3	Improper operation
4	Storage area not available
5	End of file
6	Device not working properly
7	Specified file does not exist

Appendix B. Extended BASIC Error & Warning Messages

When TI Extended BASIC is used with the TI-99/4A computer, a unique set of error and warning messages will be generated when a problem occurs. Error messages are often generated when a command or statement is entered. However, additional error situations may occur during the scanning procedure or during the actual execution of the program.

If an error message was displayed when a statement or command was entered, the error message will only apply to the most recently entered statement. Errors that occur during the scanning or execution of a program will be displayed along with the number of the program line containing the problem.

Bad Argument

This error will occur when a function has an inappropriate value as an argument.

Bad Line Number

This error will occur when a statement specifies a line number that does not exist in a program. Bad Line Number errors generally occur in the following types of statements: GOTO, GOSUB, IF-THEN, etc.

Bad Subscript

This error will occur when the subscript of an array is a non-integer or lies outside of the allowable range of values.

Bad Value

This error message indicates that a statement contains one or more inappropirate values. The parameters in many statements are restricted to specific types of values. Values outside of the allowable range will cause a Bad Value error.

Can't Continue

This error will result when the CON command is used improperly. This command can only be used if a breakpoint in the

execution of a program has occurred. The CON command cannot be used if a program has been halted due to an error or if a program is being edited after having been halted.

Command Illegal in Program

This error results from the use of one of the following commands in a program: BYE, CON, LIST, MERGE, NEW, NUM, OLD, RES, or SAVE.

Data Error

A Data Error is the result of a problem with a DATA, READ, or RESTORE statement. A program must be organized in such a way that each variable in each READ statement corresponds to a valid data value.

File Error

A File Error will occur when an inappropriate statement has been executed for a data file. For example, an OPEN statement for a file that is already open will cause an error. Similarly, a CLOSE statement for a file that has not been opened will also cause an error.

For-Next Nesting

Each FOR statement in a program must have a corresponding NEXT statement. These loops must be arranged so that each loop is fully contained in another loop, or so that each loop has been terminated before another begins.

I/O Error

This type of error can be caused by a wide range of input or output operations. This error message is unique in the fact that it is accompanied by a two digit code that describes the exact nature of the error condition. The first digit of the code specifies the type of statement causing the error condition.

first digit	statement
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

The second digit of the code describes the specific nature of the problem.

second digit	description
0	Device not present
1	Write Protection in effect
2	Improper file specification
3	Improper operation
4	Storage area not available
5	End of file
6	Device not working properly
7	Specified file does not exist

Illegal After Subprogram

A SUBEND statement cannot be followed by an END, REM, or SUB statement.

Image Error

An IMAGE, PRINT USING, or DISPLAY USING statement must not specify more than 14 significant figures for a decimal value or more than 10 figures for a value in scientific notation. Also, a format string must not contain more than 254 characters.

Improperly Used Name

This error message will be displayed when a variable, array or function name has been misused. None of the TI Extended BASIC reserved words can be used as a variable, array or function name. Also, the same name cannot be used for a variable, array or function name simultaneously. This error also will occur when an array has been dimensioned twice or when a subscripted variable has been used in a FOR statement.

Incorrect Argument List

This error will occur when CALL and SUB statements for a subroutine do not have corresponding arguments.

Input Error

When data is being entered from the keyboard, an Input Error will merely cause a warning message. However, an Input Error caused by a data file operation will cause program execution to be terminated. INPUT requires that a value be input for each variable specified with the statement. Also, the type of data being input must correspond to the type of each variable indicated with INPUT. If either of these conditions are violated, an Input Error occurs.

Line Not Found

A Line Not Found error will occur when a statement or command specifies a line number that does not exist in a program.

Line Too Long

This error will occur when a program line contains too-many characters.

Memory Full

A Memory Full error will result when an insufficient amount of the computer's memory is available to perform the specified command or accommodate the specified program line.

Missing SUBEND

This error will occur when a SUBEND statement does not appear as required in a program.

Must Be In Subprogram

This error will occur when a SUBEND or SUBEXIT statement appears in an inappropriate location in a program.

Name Too Long

A Name Too Long error will occur when a variable name contains more than 15 characters.

Next Without For

This error will occur when a FOR statement is missing, or when the structure of a loop is incorrect.

No Program Present

A program must be present in the computer's memory before a RUN, LIST, SAVE, RESTORE, or RESEQUENCE command can be executed.

Numeric Overflow

A Numeric Overflow warning message will be displayed if a number in scientific notation requires an exponent greater than 127. This warning condition does not terminate program execution.

Only Legal in a Program

Many statements can only be used in the program mode. This error will occur if one of these statements has been entered as a command.

Option Base Error

This error will occur if more than one OPTION BASE statement appears in a program. This error will also occur if an OPTION BASE statement specifies a value other than zero or one.

Protection Violation

This error will occur if an attempt is made to save, list, or edit a program that had previously been saved as a protected program.

Recursive Subprogram Call

This error will result when a subprogram contains a reference to itself.

Return Without Gosub

A RETURN Without GOSUB error will occur when a RETURN statement is encountered in a program without a corresponding GOSUB statement.

Speech String Too Long

This error will occur when the SPGET subprogram returns a string expression in excess of 255 characters.

Stack Overflow

A Stack Overflow error will occur when an expression contains too many sets of parentheses. This error will also occur if an insufficient amount of memory is available to perform a set of calculations.

String Truncated

A String Truncated warning message will be displayed when a string expression must be reduced in length in order to be within the 255 character limit. The execution of the program will not be terminated when this condition occurs. However, any extra characters in the string expression will be abolished.

String-Number Mismatch

This type of error will occur whenever a string value was used instead of a numeric value, or vice versa.

Subprogram Not Found

This error will occur when a CALL statement is used in an attempt to access a non-existent subprogram.

Syntax Error

Syntax Errors result from statements or commands which have been entered with an incorrect format. Misspelled keywords and the incorrect use of punctuation marks are the most common sources of syntax errors.

Unmatched Quotes

This type of error will occur when a command or statement contains an odd number of quotation marks.

Unrecognized Character

An Unrecognized Character error will occur when a special character has been included in a statement without having been enclosed in quotation marks.

Appendix C. ASCII Codes

The TI-99/4A computer uses special numeric values to represent each character. These character codes are used in CHR$, VCHAR, HCHAR, and SPRITE statements as well as many others.

The lowercase characters have the same shapes as the uppercase characters, but the lowercase letters are smaller than the uppercase letters.

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
32	(space)	52	4	72	H
33	!	53	5	73	I
34	"	54	6	74	J
35	#	55	7	75	K
36	$	56	8	76	L
37	%	57	9	77	M
38	&	58	:	78	N
39	'	59	;	79	O
40	(	60	<	80	P
41	)	61	=	81	Q
42	*	62	>	82	R
43	+	63	?	83	S
44	,	64	@	84	T
45	-	65	A	85	U
46	.	66	B	86	V
47	/	67	C	87	W
48	0	68	D	88	X
49	1	69	E	89	Y
50	2	70	F	90	Z
51	3	71	G	91	[

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
92	\	104	h	116	t
93	]	105	i	117	u
94	^	106	j	118	v
95	_	107	k	119	w
96	`	108	l	120	x
97	a	109	m	121	y
98	b	110	n	122	z
99	c	111	o	123	{
100	d	112	p	124	|
101	e	113	q	125	}
102	f	114	r	126	~
103	g	115	s		

Appendix D. TI BASIC Commands, Functions, & Statements

Table D-1. TI BASIC Functions

ABS	INT	SIN
ASC	LEN	SQR
ATN	LOG	STR$
CHR$	POS	TAB
COS	RND	TAN
EOF	SEG$	VAL
EXP	SGN	

Table D-2. BASIC Commands

BREAK	CONTINUE	PRINT#
BYE	DELETE	RANDOMIZE
CALL CHAR	DIM	REM
CALL CLEAR	DISPLAY	RES
CALL COLOR	EDIT	RESEQUENCE
CALL GCHAR	END	RESTORE
CALL HCHAR	LET	RUN
CALL JOYST	LIST	SAVE
CALL KEY	NEW	STOP
CALL SCREEN	NUM	TRACE
CALL SOUND	NUMBER	UNBREAK
CALL VCHAR	OLD	UNTRACE
CLOSE	OPEN	
CON	PRINT	

Table D-3. TI BASIC Statements

BREAK	DEF	ON GOTO
CALL CHAR	DIM	OPEN
CALL CLEAR	DISPLAY	OPTION BASE
CALL COLOR	ELSE	PRINT
CALL GCHAR	END	PRINT#
CALL HCHAR	GOSUB	RANDOMIZE
CALL JOYST	GOTO	READ
CALL KEY	IF THEN	REM
CALL SCREEN	INPUT	RESTORE
CALL SOUND	INPUT#	RETURN
CALL VCHAR	LET	STEP
CLOSE	NEXT	STOP
DATA	ON GOSUB	UNTRACE

Appendix E. Extended BASIC Commands, Functions, and Statements

Table E-1. Extended BASIC Functions

ABS	MAX	SGN
ASC	MIN	SIN
ATN	PI	SQR
CHR$	POS	STR$
EOF	REC	TAB
EXP	RND	TAN
INT	RPT$	VAL
LEN	SEG$	

Table E-2. Extended BASIC Commands

ACCEPT	CALL KEY	CON	RANDOMIZE
BREAK	CALL LINK	CONTINUE	READ
BYE	CALL LOAD	DELETE	REM
CALL CHAR	CALL MAGNIFY	DIM	RES
CALL CHARPAT	CALL MOTION	DISPLAY	RESEQUENCE
CALL CHARSET	CALL PATTERN	FOR	RESTORE
CALL CLEAR	CALL PEEK	LET	RUN
CALL COINC	CALL POSITION	LIST	SAVE
CALL COLOR	CALL SAY	MERGE	SIZE
CALL DELSPRITE	CALL SCREEN	NEXT	STOP
CALL DISTANCE	CALL SOUND	NUM	TRACE
CALL ERR	CALL SPGET	NUMBER	UNBREAK
CALL GCHAR	CALL SPRITE	OLD	UNTRACE
CALL HCHAR	CALL VCHAR	OPEN	
CALL INIT	CALL VERSION	PRINT	
CALL JOYST	CLOSE	PRINT#	

Table E-3. Extended BASIC Statements

ACCEPT	CALL LOCATE	END	ON GOTO
BREAK	CALL MAGNIFY	FOR	ON WARNING
CALL	CALL MOTION	GO SUB	OPEN
CALL CHAR	CALL PATTERN	GO TO	OPTION BASE
CALL CHARPAT	CALL PEEK	GOSUB	PRINT
CALL CHARSET	CALL POSITION	GOTO	PRINT#
CALL CLEAR	CALL SAY	IF THEN	RANDOMIZE
CALL COINC	CALL SCREEN	IMAGE	READ
CALL COLOR	CALL SOUND	INPUT	REM
CALL DELSPRITE	CALL SPGET	INPUT#	RESTORE
CALL DISTANCE	CALL SPRITE	LET	RETURN
CALL ERR	CALL VCHAR	LINPUT	RUN
CALL HCHAR	CLOSE	NEXT	SUB
CALL INIT	DATA	ON BREAK	SUBEND
CALL JOYST	DEF	ON ERROR	SUBEXIT
CALL KEY	DELETE	ON GO SUB	TRACE
CALL LINK	DIM	ON GO TO	UNBREAK
CALL LOAD	DISPLAY	ON GOSUB	UNTRACE

Appendix F. RESIDENT VOCABULARY

The Solid State Speech Synthesizer, when used with Extended BASIC, can recognize only a limited number of words and phrases. The following words can be used in CALL SAY or CALL SPGET statements.

0
1
2
3
4
5
6
7
8
9
A[1]
A1[2]
ABOUT
AFTER
AGAIN
ALL
AM
AN
AND
ANSWER
ANY
ARE
AS
ASSUME
AT

B
BACK
BASE
BE
BETWEEN
BLACK
BLUE
BOTH
BOTTOM
BUT
BUY
BY
BYE

C
CAN
CASSETTE
CENTER
CHECK
CHOICE
CLEAR
COLOR
COME
COMES
COMMA
COMMAND
COMPLETE
COMPLETED
COMPUTER
CONNECTED
CONSOLE
CORRECT
COURSE
CYAN

D
DATA
DECIDE
DEVICE
DID
DIFFERENT
DISKETTE
DO
DOES
DOING
DONE
DOUBLE
DOWN
DRAW
DRAWING

E
EACH
EIGHT
EIGHTY
ELEVEN
ELSE
END
ENDS
ENTER
ERROR
EXACTLY
EYE

F
FIFTEEN
FIFTY
FIGURE
FIND
FINE
FINISH
FINISHED
FIRST
FIT
FIVE
FOR
FORTY
FOUR
FOURTEEN
FOURTH
FROM
FRONT

G
GAMES
GET
GETTING
GIVE
GIVES
GO
GOES
GOING
GOOD
GOOD WORK
GOODBYE
GOT
GRAY
GREEN
GUESS

H
HAD
HAND
HANDHELD UNIT
HAS
HAVE
HEAD
HEAR
HELLO
HELP
HERE
HIGHER
HIT
HOME
HOW
HUNDRED
HURRY

I
I WIN
IF
IN
INCH
INCHES
INSTRUCTION
INSTRUCTIONS
IS
IT

J
JOYSTICK
JUST

K
KEY
KEYBOARD
KNOW

L
LARGE
LARGER
LARGEST
LAST
LEARN
LEFT
LESS
LET
LIKE
LIKES
LINE
LOAD
LONG

LOOK
LOOKS
LOWER

M
MADE
MAGENTA
MAKE
ME
MEAN
MEMORY
MESSAGE
MESSAGES
MIDDLE
MIGHT
MODULE
MORE
MOST
MOVE
MUST

N
NAME
NEAR
NEED
NEGATIVE
NEXT
NICE TRY
NINE
NINETY
NO
NOT
NOW
NUMBER

O
OF
OFF
OH
ON
ONE
ONLY
OR
ORDER
OTHER
OUT
OVER

P
PART
PARTNER
PARTS
PERIOD
PLAY
PLAYS
PLEASE
POINT
POSITION
POSITIVE
PRESS
PRINT
PRINTER
PROBLEM
PROBLEMS
PROGRAM
PUT
PUTTING

Q

R
RANDOMLY
READ[3]
READ1[4]
READY TO START
RECORDER
RED
REFER
REMEMBER
RETURN
REWIND
RIGHT
ROUND

S
SAID
SAVE
SAY
SAYS
SCREEN
SECOND
SEE
SEES
SET
SEVEN
SEVENTY
SHAPE
SHAPES
SHIFT
SHORT
SHORTER
SHOULD
SIDE
SIDES
SIX
SIXTY
SMALL
SMALLER
SMALLEST
SO
SOME
SORRY
SPACE
SPACES
SPELL
SQUARE
START
STEP
STOP
SUM
SUPPOSED
SUPPOSED TO
SURE

T
TAKE
TEEN
TELL
TEN
TEXAS INSTRUMENTS
THAN
THAT
THAT IS INCORRECT
THAT IS RIGHT
THE[5]
THE1[6]
THEIR
THEN
THERE
THESE
THEY
THING
THINGS
THINK
THIRD
THIRTEEN
THIRTY
THIS
THREE
THREW
THROUGH
TIME
TO
TOGETHER
TONE
TOO
TOP
TRY
TRY AGAIN
TURN
TWELVE
TWENTY
TWO
TYPE

U
UHOH
UNDER
UNDERSTAND
UNTIL
UP
UPPER
USE

V
VARY
VERY

W
WAIT
WANT
WANTS
WAY
WE
WEIGH
WEIGHT
WELL
WERE
WHAT
WHAT WAS THAT
WHEN
WHERE
WHICH
WHITE
WHO
WHY
WILL
WITH
WON
WORD
WORDS
WORK
WORKING
WRITE

X

Y
YELLOW
YES
YET
YOU
YOU WIN
YOUR

Z
ZERO

+[7]
-[8]
.[9]

[1] pronounced like the a in ape
[2] pronounced like the a in among
[3] pronounced reed
[4] pronounced red
[5] pronounced thee
[6] pronounced tha
[7] pronounced as the word "negative" when used before numbers.
[8] pronounced as the word "positive" when used before numbers.
[9] pronounced as the word "point" when used between numbers.

INDEX

Other Computer Titles from Weber Systems, Inc.

Compaq User's Handbook
ISBN 0-938862-11-1

Coleco Adam User's Handbook
ISBN 0-938862-45-6

Lotus 1-2-3 User's Handbook
ISBN 0-938862-54-5

IBM PCjr For Students
ISBN 0-938862-25-1

Coleco Adam For Students
ISBN 0-938862-42-1

TK Solver User's Handbook
ISBN 0-938862-00-6

Atari XL User's Handbook
ISBN 0-938862-08-1

Kaypro BASIC Programs For Business
ISBN 0-938862-51-0

Xenix User's Handbook
ISBN 0-938862-44-8

C Language User's Handbook
ISBN 0-938862-56-1

CP/M Simplified
ISBN 0-938862-04-9

CBASIC Simplified
ISBN 0-938862-10-3

User's Handbook to the Atari 400/800
ISBN 0-938862-15-4

User's Handbook to the TRS-80 Model II
ISBN 0-938862-01-4

User's Handbook to the Apple II
ISBN 0-938862-03-0

Please Contact your local bookstore or computer store to order.
For catalogue, send a self addressed stamped envelope to:

Weber Systems Inc.
Box 413 Gates Mills, OH 44040

Updates and Revisions

Due to the changing nature of the personal computer market, Weber Systems, Inc. regularly updates and revises its titles. If you wish to receive notification of the availability of revised editions of the TI-99/4A User's Handbook, please return this page to the following address:

Weber Systems, Inc.
Box 413
Gates Mills, Ohio 44040

Also, we would appreciate any comments you might have on this title.

Please send notification of updates in
TI-99/4A User's Handbook to:

Name _____

Address _____

City _____ State _____ Zip _____

Comments _____

_____

_____

_____

_____

_____

_____

_____

$14.95

TI-99/4A® USER'S HANDBOOK

The TI-99/4A® computer has impressive computing, graphics, speech, and sound generating capabilities. The **TI-99/4A® User's Handbook** provides clear, concise, and complete instructions which allow the user to master these capabilities.

The **TI-99/4A® User's Handbook** is written in a simple, concise manner so that even a first-time user can master the TI-99/4A®, yet it still contains a wealth of advanced information for the experienced user. A complete guide to the operation and programming of the TI-99/4A® computer and its related peripheral equipment is included.

The following topics are covered in detail:

- TI BASIC® Programming Techniques
- Extended BASIC® Programming Features
- Programming Sound and Graphics
- Reference Guide to Both Versions of BASIC
- Using the TI Disk Memory System®
- Using the TI Program Recorder®
- Using the TI-99/4A® Printer

The **TI-99/4A® User's Handbook** is a must for any TI-99/4A® user or potential user.

LC: 83-060591

ISBN: 0-938862-49-9