Texas Instruments Home Computer

User's Reference Guide

Quick Guide to TI BASIC

IMPORTANT

Record the serial numbers and purchase dates of the TI Home Computer and Color Monitor in the space below. The serial number is identified by the words "SERIAL NO." on the units. Always reference this information in any correspondence.

TI Home Computer
Model No.__________ __________ Serial No. __________ Purchase Date

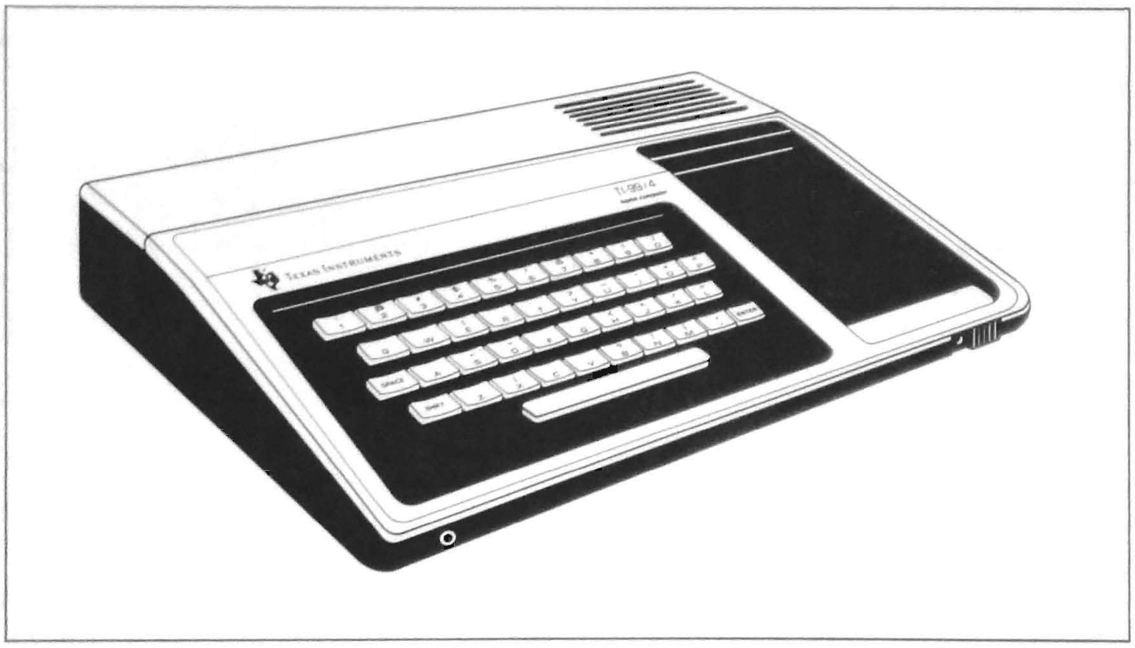
TI Color Monitor __________ Serial No. __________ Purchase Date

TEXAS INSTRUMENTS HOME COMPUTER

LCB-4491

User's Reference Guide

A complete, detailed guide to using
and enjoying your Texas Instruments Home Computer.



WARNING

THIS EQUIPMENT HAS BEEN CERTIFIED TO COMPLY WITH THE LIMITS FOR A CLASS B COMPUTING DEVICE, PURSUANT TO SUBPART J OF PART 15 OF FCC RULES. ONLY PERIPHERALS (COMPUTER INPUT/OUTPUT DEVICES, TERMINALS, PRINTERS, ETC.) CERTIFIED TO COMPLY WITH THE CLASS B LIMITS MAY BE ATTACHED TO THIS COMPUTER. OPERATION WITH NON-CERTIFIED PERIPHERALS IS LIKELY TO RESULT IN INTERFERENCE TO RADIO AND TV RECEPTION.

1040003-1
(USE WITH 1015963-1)

See important warranty information at back of book.

This book was developed by:

The Staff of the
Texas Instruments Learning Center

and

The Staff of the
Texas Instruments Personal
Computer Division

Artwork and layout
were coordinated and executed by:

Schenck, Plunk & Deason

ISBN# 0-89512-029-1

Library of Congress Catalog #79-65511

Table of Contents

I. GENERAL INFORMATION

II. EQUATION CALCULATOR

Table of Contents

Table of Contents

Table of Contents

General Information

INTRODUCTION

You are about to be introduced to the exciting new world of the home computer. Until just a few years ago, the size, price, and complexity of computers put them beyond the reach of the individual purchaser. Today, the Texas Instruments Home Computer brings you remarkable computing power in an affordable, compact unit that can be easily set up in your home or office.

Whether you have years of computer experience or have never worked with computers before, the innovative and flexible features of your Home Computer offer you a wide variety of applications. Within minutes, using the special built-in features or convenient preprogrammed modules, you can begin using your computer to

- manage your personal resources
- develop projects for home and business
- bring new dimensions to education — for you and your children
- provide engaging new types of entertainment for the entire family
- and much, much more.

Powerful Built-In Features

Your Texas Instruments Home Computer is designed with two powerful features built right in. They are:

TI BASIC

This feature makes your TI Home Computer a "true" computer — not a video game or electronic toy. Using a simple but very powerful computer language, TI BASIC, you can develop and use your own computer programs for applications ranging from color graphics to statistical analysis and more.

EQUATION CALCULATOR

This innovative feature gives you versatile computing power in an easy-to-use video calculator form. You can use the Equation Calculator for problems ranging from straightforward addition and subtraction to evaluating more complex formulas and equations.

Convenient Module System

The unique system of easy-to-use, snap-in *Solid State Software*™ Command Modules* assures the continued versatility and usefulness of your computer. These rugged, all solid state modules are completely preprogrammed for you. You just snap them in, and they "prompt" you through activities, applications, games, entertainment, etc. With a module plugged into the computer console, you can start using your computer immediately. You can choose from a wide selection of Command Module titles. Ask your dealer to see all of them!

USING THIS BOOK

This *User's Reference Guide* is one of the three books packed with your computer. It tells you everything you need to know to fully enjoy all the features and functions of your computer. The *Read This First* booklet included with the computer covers first things first: what is packed with your computer, initial set-up, getting started using the Command Module software. The *Beginner's BASIC* book that is also included is a self-paced and very enjoyable short course in BASIC programming.

This *User's Reference Guide* is organized in a step-by-step fashion:

- a brief discussion of the placement and care of your new computer
- a tour of your computer, starting with the connector outlets and including the computer keyboard
- the use of the accessories available for the computer
- sections on the EQUATION CALCULATOR and TI BASIC

A final note before we begin. You don't need a technical background to fully enjoy and utilize your Home Computer. No special expertise or experience is necessary. The simple instructions we'll provide here and in the books provided with each module, as well as the prompting you'll receive from the computer, are all you'll need to get "up and running" quickly.

*sold separately

General Information

PLACEMENT AND CARE

With the proper care, your Texas Instruments Home Computer will provide you with many years of enriching experiences. Treat your computer with the same good care you'd give other electronic products such as a television or stereo.

It's important that you choose a good location for your computer system. Select a place for the system where sunlight or bright light doesn't fall directly on the screen. DO NOT SET THE COMPUTER CONSOLE ON THE TOP OF A TELEVISION SET.

Correct ventilation is necessary for the continued proper operation of your computer system. Be sure air can flow freely through all the ventilation slots on the bottoms, backs, and tops of the console and the monitor. Do not obstruct the ventilation or enclose the system in any way. It's best to place the system on a hard-topped non-metallic surface such as a table.

From time to time you may want to clean the surfaces of your computer. First, turn the computer OFF. Then gently wipe the surface using a damp, lint-free cloth. Do not use solvents or other cleansers to clean the computer console.

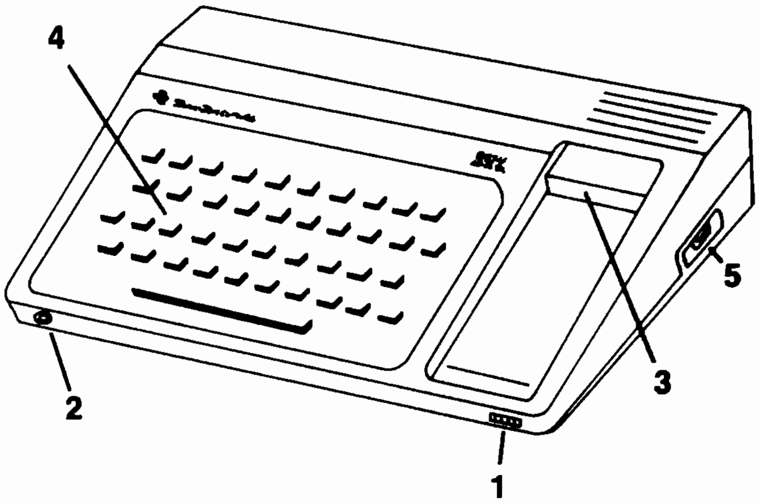
CAUTION: *Electronic equipment can be damaged by static electricity discharges.* Static electricity build-ups can be caused by walking across a carpet. If you build up a static charge and then touch the computer, a Command Module, or any accessory device, you can permanently damage the internal circuits. Always touch a metal object (such as a door knob, a desk lamp, etc.) before working with your computer, connecting accessory devices, handling a Command Module, inserting a Command Module, etc. You may want to purchase a special anti-static spray for use on the carpeting in the room where your computer is located. This commercial preparation is usually available from local carpet, hardware, and office supply stores.

A TOUR OF YOUR COMPUTER

Your computer console is the central part of your computer system. It's designed so that all of the other units of the system easily connect to this console. No tools are required. Let's-become more familiar with the console by identifying the various outlets, and then tour the keyboard.

Getting Started

Let's look at the front and right side of your computer.



1 This is the ON/OFF switch. The small light next to the switch indicates when the computer is ON.

2 This outlet provides for the connection of earphones or a headset.*

3 Command Module software* snaps into this outlet. (See pages 13-14 for more details.)

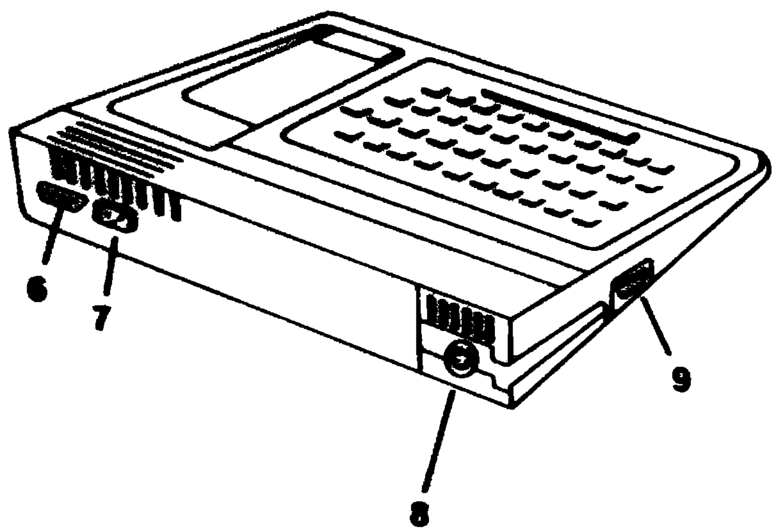
4 This keyboard is used to type information into the computer.

5 This outlet is for optional peripheral accessories. Details are included with the appropriate peripheral.

*sold separately

General Information

This is the back and left side of the console:



6 The Cassette Interface Cable* connects to the console at this 9-pin "D" outlet. (See pages 14-16 for more details.)

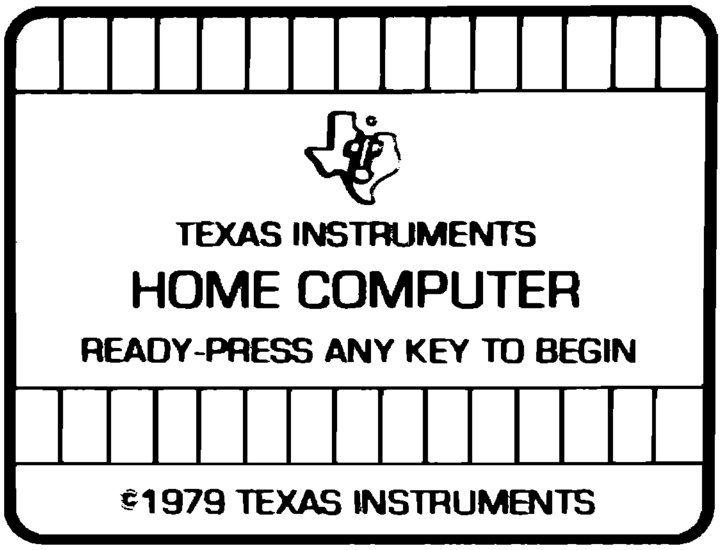
7 The Power Cord attaches to the console at this 4-pin outlet. (See *Read This First* or page 200 of this book for complete details.)

8 This 5-pin connector (also called a DIN connector) is for audio-out and video-out. This connector will insert easily when properly aligned. (See *Read This First* or page 199 of this book for complete details.)

9 The Wired Remote Controllers connect to this 9-pin outlet. Details are included with the accessory.

(*Note*: Do not confuse this 9-pin outlet with the 9-pin outlet on the back of the console. They are not interchangeable.)

Turn on the computer following the directions in *Read This First*. You'll first see the master title screen.



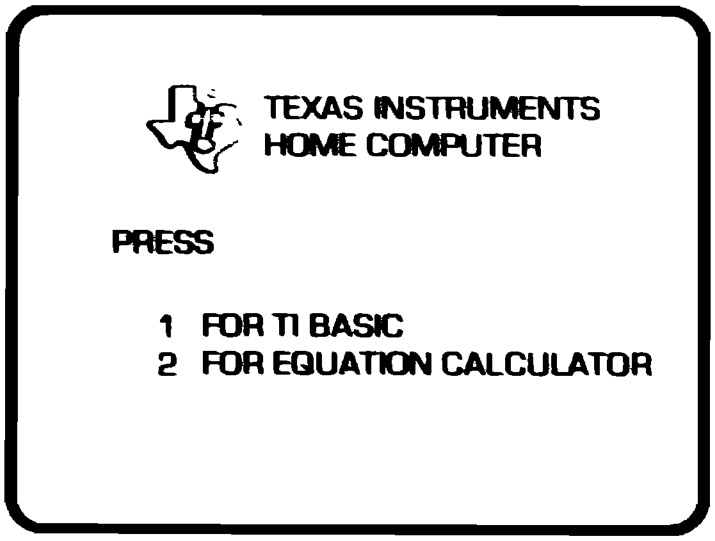
TEXAS INSTRUMENTS

HOME COMPUTER

READY-PRESS ANY KEY TO BEGIN

©1979 TEXAS INSTRUMENTS

Next, press any key on the keyboard. You'll see a screen called the master selection list.



TEXAS INSTRUMENTS
HOME COMPUTER

PRESS

1 FOR TI BASIC
2 FOR EQUATION CALCULATOR

Let's press **1** for TI BASIC, and try out some of the keys as we go along.

The small flashing symbol you see in the lower left corner of the screen is called the *cursor*. This symbol shows you exactly where the next character (letter, number, punctuation, etc.) will appear on the screen when you press a key.

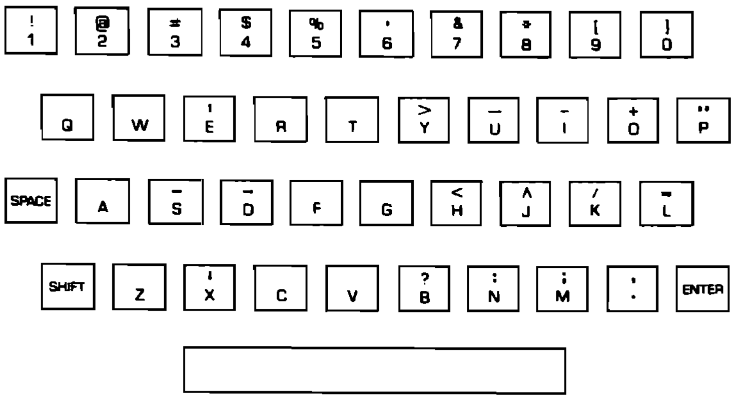
(*Note*: If you should accidentally leave your computer on for a long period of time, the screen will automatically go blank after about ten minutes of non-use. Press any key to bring the display back again.)

*sold separately

General Information

A Tour of the Keyboard

Let's take a close look at the keyboard.



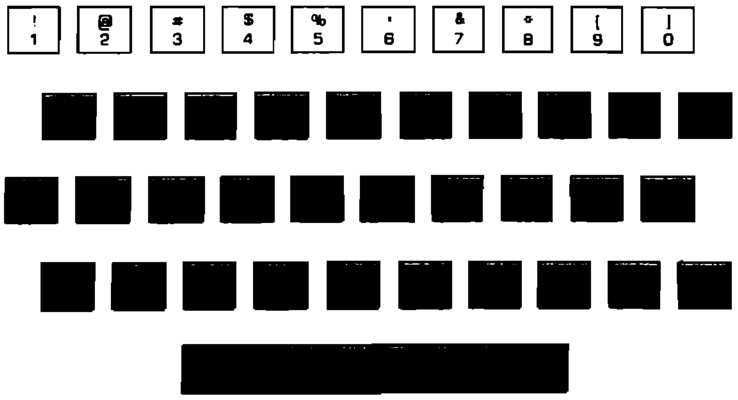
The keyboard is quite similar to that of a standard typewriter, with keys of several types. Some have special uses, while others perform simple, straightforward functions. We'll discuss each group of keys separately.

ALPHABET KEYS

All alphabetical symbols are typed into the computer using the alphabet keys. You don't have to be concerned about capitalizing any letters because the computer uses only *upper case* (or capital) letters.

NUMBER KEYS

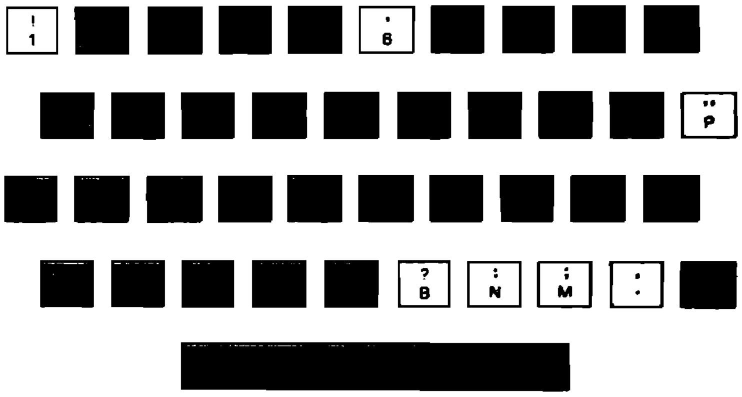
The number keys are located on the top row of your computer keyboard.



These keys are used to type numbers into your computer. If you have previous typing experience, you need to be aware of two differences between this keyboard and some typewriter keyboards. With this computer, you cannot type the letter "L" as the number "1." Also, never substitute the letter "O" for a zero. The computer screen displays the letter "O" with squared corners and displays a zero with rounded corners, so you'll be able to distinguish them.

PUNCTUATION KEYS

The following keys are used for punctuation:



All punctuation keys (except the period) require that the **SHIFT** key be pressed and held while the punctuation key is pressed. For example, to display a question mark, simply hold down the **SHIFT** key and press **?**. We'll indicate the use of the **SHIFT** key as follows: **SHIFT ?**.

SPECIAL FUNCTION KEYS

Several keys have varying functions in TI BASIC, the Equation Calculator, some Command Module software and other applications. The exact use of the keys is described in detail in the appropriate sections of this *Guide* or in the manuals that accompany the various modules. Let's become familiar with these keys.

SHIFT Q (QUIT)

Pressing SHIFT Q (at any time) returns the computer to the Master Title Screen. *Note:* **When you press SHIFT Q all data or program material you have entered will be erased.**

ENTER (FWD)

In most cases, when you press the orange **ENTER** key, you tell the computer to accept the information you have just finished typing. Additional functions will be explained in the appropriate manuals.

SHIFT [←] (LEFT)

Pressing the *left-arrow* key (backspace) moves the cursor one space to the left each time it is pressed. The cursor does not erase or change the characters on the screen as it passes over them.

SHIFT [→] (RIGHT)

Pressing the *right-arrow* key (forwardspace) moves the cursor one space to the right each time it is pressed. As the cursor passes over the characters printed on the screen, it does not alter them in any way.

SHIFT F (DEL)

The *delete* key is used to delete a letter, number, or other character from the line you have just typed (before you press **ENTER**).

SHIFT G (INS)

The *insert* key is used to insert a letter, number, or other character into the information you have just typed (before pressing **ENTER**).

SHIFT [↑] (UP)
SHIFT [↓] (DOWN)

These keys have various functions according to the specific application where they are used. See the sections on TI BASIC and the Equation Calculator (pages 21, 33 and 66 in this book) and the appropriate module manuals for a complete explanation of their use.

SHIFT C (CLEAR)

This key is normally used to clear from the screen any information you have typed (before pressing **ENTER**). It also has additional functions in TI BASIC. See page 34 of this book for details of its use in TI BASIC.

SHIFT T (ERASE)

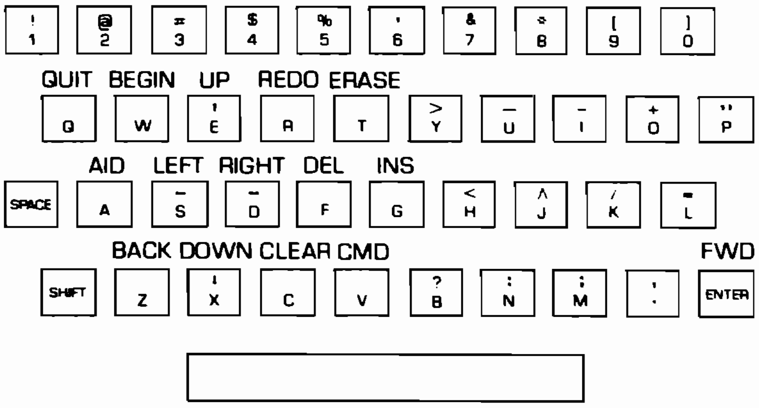
This key also is used to clear the line you are presently typing (before you press **ENTER**). See page 34 of this book for an explanation of its function in TI BASIC.

Other keys have special functions in the various modules. Some of these are:

SHIFT W (BEGIN)
SHIFT R (REDO)
SHIFT A (AID)
SHIFT Z (BACK)
SHIFT V (CMD)

Keyboard Overlay

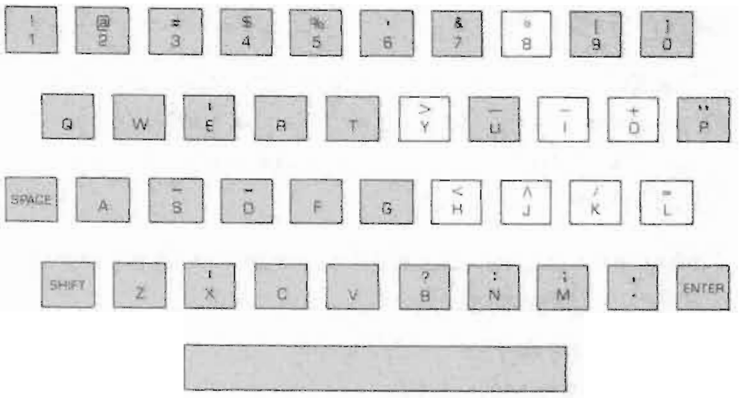
A plastic keyboard overlay is included with your computer. You can use this overlay to help you more easily identify the functions of certain keys that are used in combination with the **SHIFT** key.



The overlay reminds you that you can press **SHIFT A** for "AID," **SHIFT Q** for "QUIT," etc. The function of the **SHIFT V** key (labelled "CMD") is defined in the modules or applications where this key is used.

MATH OR OPERATION KEYS

The Math keys (or operation keys) are the keys used to instruct the computer to add, subtract, multiply, divide, and raise a number to a power.



The symbols for addition, subtraction, and equals are the usual ones you're familiar with, but the multiplication and division symbols may be new to you.

+	Addition
−	Subtraction
*	Multiplication
/	Division
=	Equals

The "caret" key (Λ) is also used for mathematical operations:

SHIFT Λ
This symbol tells the computer to perform exponentiation (raising a number to a power). Since 5^3 cannot be easily printed on your screen, the computer interprets 5 Λ 3 to mean that three is an exponent.

The following keys are used to indicate mathematical relationships in TI BASIC:

SHIFT >	"Greater than"; this symbol is used to compare two quantities.
SHIFT <	"Less than"; this symbol is also used to compare two quantities.

Notice that you must press the **SHIFT** key *and* then the operation key to type in any of these symbols. (You can perform arithmetic in both TI BASIC and the Equation Calculator. Mathematical operations and examples are discussed in the appropriate sections of this book.)

SPACE BAR

The **SPACE BAR** is the long blank bar at the bottom of the keyboard. It operates just like the space bar on a regular typewriter. When you press the **SPACE BAR**, the computer leaves a blank space between words, letters or numbers.

The **SPACE BAR** can also be used to erase characters already on the screen. (See the section titled "Correcting Errors" on page 13.)

The **SPACE** key has the same functions as the **SPACE BAR**.

OTHER SYMBOLS

This keyboard also includes the usual special symbols of a typewriter keyboard. These symbols require that you press and hold the **SHIFT** key while pressing another key. (*Note:* **SHIFT** ___ cannot be used to underline because it will erase the original characters.)

Correcting Errors

If you want to correct a typing error before you press **ENTER**, you can do it easily. Move the cursor back to the character you want to change (using **SHIFT** ←). Retype the correct character (or characters); then move the cursor back to the end of the word or phrase you were typing (using **SHIFT** →).

You can erase errors by using the **SPACE BAR**. Backspace (using **SHIFT** ←) to a point where you want to begin erasing. Then press the **SPACE BAR** to move the cursor over the characters on the screen. The characters are erased.

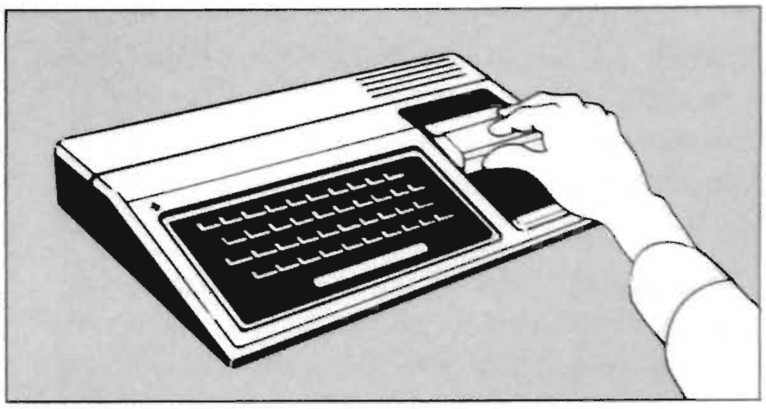
In certain applications you can also make corrections using the **SHIFT F** (delete) key and the **SHIFT G** (insert) key.

ACCESSORIES

Several accessories are available for use with the computer. These accessories expand the capabilities of your basic unit and enable you to derive greater benefit and enjoyment from your computer. This flexibility to increase the computer's capability lets you build your system as you need it. Some of these accessories are discussed separately below.

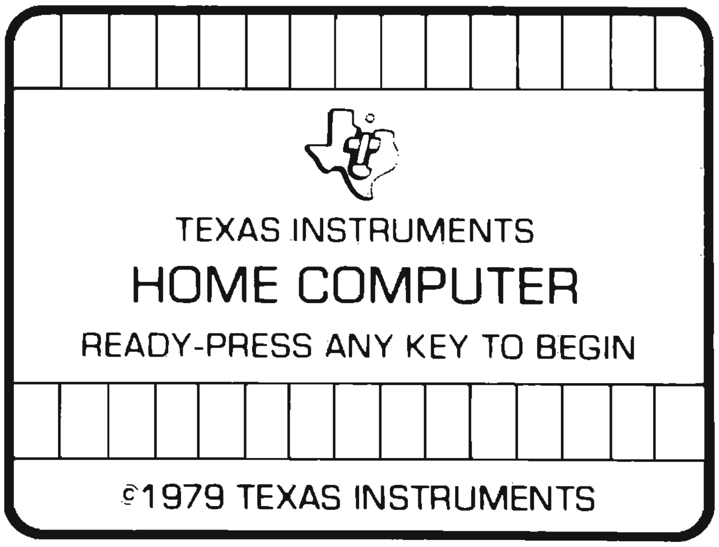
Solid State Software® Command Modules

The Texas Instruments Command Module software system provides you with solid state modules preprogrammed with activities and applications that range from home management to education to entertainment. The modules are easy to use — they prompt and help you through the activities. Each module comes with its own instruction manual that explains how you interact with the computer in each activity. Because no special computer skills are required, these easy-to-use plug-in modules allow you to begin using your computer immediately. Here's how you insert a module into the computer:



Note: Before inserting a module into the computer, be sure that it has not built up a static charge. See page 8 of this *Guide*.

1. *If the computer is OFF*, slowly slide the module into the slot on the console, and turn the computer on. The master title screen will appear.



TEXAS INSTRUMENTS

HOME COMPUTER

READY-PRESS ANY KEY TO BEGIN

©1979 TEXAS INSTRUMENTS

If the computer is ON, hold down the **SHIFT** key and press **Q** to make the master title screen appear. *Then* slide the module into the slot on the console.

2. Press any key to make the master selection list appear. The title of the module you've inserted will be third on the list.
3. Press the **3** key to select the module and begin using it.

When you press key **3**, the title screen of the module shows on the screen. Follow the directions given on the screens and in the module instruction manual as you use each module.

If a module does not appear to be operating properly, return to the master title screen by pressing **SHIFT Q**. Withdraw the module and reinsert it carefully. Then press any key to make the master selection list appear. The title of the module should show up in the third position. Press **3** to select the module. (*Note:* In some instances, it may be necessary to turn the computer off, wait several seconds, and then turn it on again.)

To remove the module, *first* return the computer to the master title screen by pressing **SHIFT Q**. *Then* remove the module from the slot.

An automatic reset feature is built into the computer. Anytime a module is inserted into the console, the computer should return to the master title screen. (Note: All data or program material you have entered will be erased.) In rare instances, if the module is accidentally removed from the console while the module contents are being used, the computer may behave unpredictably. To restore the computer to normal operation, turn the computer console off. Wait a few seconds. Then reinsert the module, and turn the computer on again.

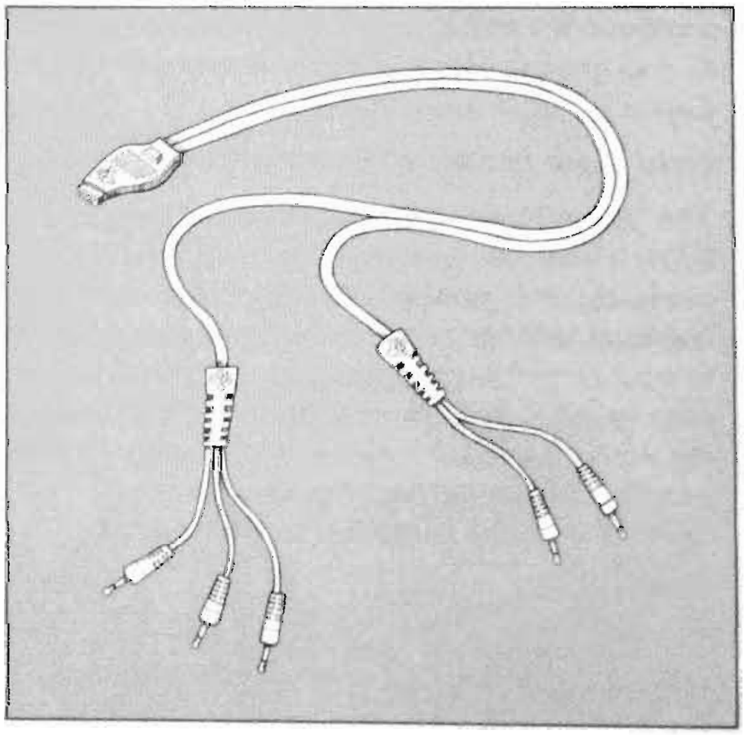
Additional information concerning use and service can be found on page 201 of this *Guide*.

A wide selection of modules is currently available and many more are on the way. This system allows you to expand and increase the capabilities of your computer as your own needs and interests change.

Cassette Interface Cable

You can further expand your computer system by using audio cassette tape recorders. TI BASIC allows you to store and retrieve data you enter in the computer (programs, numerical data, etc.). By recording data on a tape, you can *save* it as a permanent record. Later you can *load* the data from the cassette tape into the computer's memory if you want to use that information again. Several of the command modules also use this feature to save and load data you've used in the module.

You can use either one or two recorders for this purpose. Using two cassette recorders is especially helpful for advanced programming applications.



General Information

Cassette recorders are connected to the Home Computer by the special *cassette interface cable** accessory. Many standard cassette recorders can be used with the Home Computer. For best operation, however, they should have such features as:

- Volume control
- Tone control
- Microphone jack
- Remote jack
- Earphone or external speaker jack
- Digital tape counter (This will enable you to easily locate the correct tape position in case you want to store more than one program or data set on the same tape.)

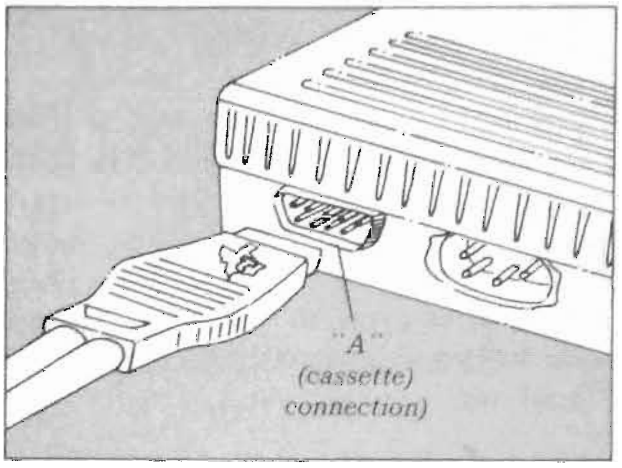
Since motor control design varies from manufacturer to manufacturer, we have tested several different cassette recorders to determine whether they can be used with the Home Computer. A list of recorders that appear to work well with this computer system is included separately. We've also indicated the volume setting and tone control setting for each unit that give the best operating results.

Texas Instruments can assume no responsibility for any design changes made by the cassette recorder manufacturers that might affect the use of a specific recorder with the TI Home Computer.

Note: The cassette interface cable uses the triple-plug end for cassette number 1 "CS1," and the double-plug end for cassette number 2 "CS2." Cassette unit 1 may be used for both recording (writing) and reading; cassette unit 2 may be used for writing only.

To connect your cassette player(s) to the Home Computer, use the cassette interface cable, and follow these simple steps:

1. Insert the single plug end of the cable with the 9-pin "D" connector into the 9-pin outlet on the back of the console (labelled "A").



2. Attach the triple plug ends into the cassette recorder(s) as follows:

- Insert the plug with the red wire into the microphone jack
- Insert the plug with the black wire into the remote jack (note that this plug is smaller than the other two plugs)
- Insert the plug with the white wire into the earphone jack (or external speaker jack) – CS1 only.

3. Make sure you notice how the cassettes are connected when you select either CS1 or CS2 when saving data. When loading data, only CS1 can be used. See TO SAVE/LOAD DATA section for more information.

(*Note:* You will usually elect to connect only one cassette recorder. The other plug end will simply be inactive when only one recorder is used with the computer.)

*Sold separately

After all cables are connected, turn the tone control on your cassette player to full TREBLE or to the point indicated on the table on the separate cassette sheet. Set the volume at about half scale (if the volume control has ten positions, set it at five or at the position indicated in the table). If your cassette player does not have a tone control, you may have to set the volume control higher for best results.

TO SAVE/LOAD DATA

If you have your cassette machine(s) connected to the console as instructed, you are ready to save/load data.

Before you attempt to save/load your data, make sure that:

- You are using *high quality* audio tape. Poor quality tape yields poor performance.
- The tape is *not longer* than C-60. Longer tapes are thinner, and provide less fidelity.
- The cassette machine is not located within two feet of the monitor or a television set to minimize magnetic field interference.
- The tape is never placed within two feet of the monitor, a television set, an electric motor, or any other strong source of magnetic fields to avoid accidental erasure of your data.
- The system (computer console, cassette machine, and Color Monitor) is not located on a continuous metallic surface to minimize conducted noise.
- You are using only CS1 for LOAD. CS1 or CS2 can be used for SAVE.

To Save/Load Data in TI BASIC

For complete instructions on how to save and load data when you are programming in TI BASIC, see pages 68-70 of this *Guide.*

To Save Data When Using a Module

After you have entered your data into the computer and connected the recorder to the computer (with a good quality tape cassette in place), you are ready to begin recording. Select the "SAVE" option offered by the module you're using. The computer asks you to press **1** for CASSETTE or **2** for OTHER DEVICE. Press **1**. Next you are asked to press **1** for CS1 (cassette unit 1) or **2** for CS2 (cassette unit 2). This time, let's press **1** and record the data on cassette unit CS1. (The instructions are the same for both CS1 and CS2.)

From this point on, the computer guides you through the SAVE routine with on-screen instructions. (The computer controls the recorder motor power. Therefore, the tape does not start to move until you press **ENTER** at the points indicated.)

Screen Instructions

* REWIND CASSETTE TAPE CS1
THEN PRESS ENTER

Procedure

Rewind the tape *before* you press **ENTER**. If your recorder *does not* have a tape-position counter, rewind the tape all the way to the beginning. If your recorder *does* have a position counter, position the tape at the spot where you want to begin recording, and press the "stop" button on the recorder. (Write down the position for later reference.) Then press **ENTER** to continue.

* PRESS CASSETTE RECORD CS1
THEN PRESS ENTER

Press the "record" button on the recorder, and then press ENTER on the computer. As soon as you do, your data will begin recording on the tape, and the screen will show this message:

* RECORDING

You may hear the sound of the encoded information as it is being stored or read from the tape unit. Several seconds of blank tape will be recorded to allow for the leader on the tape.

* PRESS CASSETTE STOP CS1
THEN PRESS ENTER

When all the data has been recorded, press the "stop" button on the recorder, and then press the computer's ENTER key.

Once you've done this, you'll be asked the following question:

* CHECK TAPE (Y OR N)?

At this point you may choose to let the computer check your tape to make sure everything was recorded properly. We strongly recommend that you do so to ensure the accuracy of your tape for future use. *Note:* CS1 only.

If you decide not to check the tape, press N for no. Remove your tape, and label it for later reference.

If you want to check the tape, press Y for yes. Again, the computer guides you with the following messages:

Screen Instructions

* REWIND CASSETTE TAPE CS1
THEN PRESS ENTER

Procedure

Rewind the tape (*before* pressing ENTER) to the point where you began recording your data. If you stored your data at the beginning of the tape, simply rewind the tape to the beginning. If, however, you began at a point other than the beginning of the tape, rewind the tape to that position, and press the "stop" button on the recorder. Then press ENTER.

* PRESS CASSETTE PLAY CS1
THEN PRESS ENTER

Press the "play" button on the recorder, and then press ENTER. The computer will compare the data in its memory to the data on the tape. While your tape is being checked by the computer, you'll see this message:

* CHECKING

If there are no errors, the following messages are displayed on the screen:

* DATA OK
PRESS CASSETTE STOP CS1
THEN PRESS ENTER

You can now remove your data tape and label it for future use.

If, however, the data were not recorded properly, you'll receive one of two error messages:

Error Message

* ERROR – NO DATA FOUND

Meaning

Your data was not recorded, or it did not play back.

PRESS R TO RECORD CS1
PRESS C TO CHECK
PRESS E TO EXIT

Error Message

* ERROR IN DATA DETECTED

Meaning

Some part of your data did not record properly.

PRESS R TO RECORD CS1
PRESS C TO CHECK
PRESS E TO EXIT

Before you go further, you may want to recheck these items:

■ Is the recorder at a proper distance from your television set (two feet or more)? ■ Is the recorder attached properly to the computer? (See pages 15-16.) ■ Is the cassette tape in good condition? (If in doubt, try another tape.) ■ Are the cassette recorder volume and tone adjusted correctly? Was the volume too high or too low? ■ Does the cassette tape head need cleaning? ■ Is the system located on a metal surface?

When you have checked these, you can choose one of these three options:

- Press **R** to record your data again, using the same instructions for RECORD that are discussed above.
- Press **C** to instruct the computer to check your data again.
- Press **E** to "exit" and the following message appears:

  * PRESS CASSETTE STOP CS1
  THEN PRESS ENTER

The "exit" key takes you back to the beginning of the "Save" option of the module. Thus, when you press **ENTER**, you see the "Save Data" screen and can try to store your data again. Just follow the instructions as they appear on the screen.

To Load Data When Using a Module

The next time you want to use the information stored on the tape, you'll need to "load" your data – that is, read the data you saved on tape *into* the memory system of the computer.* First, connect your cassette recorder(s) to your computer (see pages 15-16). Then insert into the computer the module from which you saved the information. When you're ready to "load," select the "LOAD DATA" option of the module. When the computer asks, press the **1** key to indicate the information is being read from a cassette. Then press the **1** key again to select cassette unit CS1. Remember CS1 is used for loading data.

From this point, the computer prints instructions on the screen for you to follow.

Screen Instructions

* REWIND CASSETTE TAPE CS1
THEN PRESS ENTER

Procedures

Rewind the tape *before* you press **ENTER**. Position your tape at the point from which you want to read the data into the computer (at the beginning if your recorder does not have a position counter). Then press **ENTER**.

Screen Instructions

* PRESS CASSETTE PLAY CS1
THEN PRESS ENTER

Procedures

Press the "play" button on the recorder and the **ENTER** key on the computer. The information is read from the tape and entered into the computer's memory. While the computer is reading the tape, the following message appears on the screen:

* READING

It takes some time to read in the data, depending on the amount of information stored. When the computer finishes reading the data, it tells you whether or not it read the data properly. If the data was read correctly, you'll see the following messages on the screen:

* DATA OK
* PRESS CASSETTE STOP CS1
THEN PRESS ENTER

You're now ready to begin working with the module.

If, however, the data has not been entered properly into the computer's memory, you'll see one of several "error" messages. Follow the directions on the screen to try to load your data again.

If you still have difficulty, you'll want to make sure:

- you are loading the correct tape
- the tape is positioned at the correct starting place for the data you are loading
- the tape has not been damaged or accidentally erased
- the recorder is a proper distance from your television set (two feet or more)
- the recorder is attached properly to the computer
- the cassette recorder volume is adjusted correctly
- the system is not located on a metal surface
- the tape was recorded with your cassette unit or an identical model
- the cassette tape head is clean
- you are using cassette unit 1

**Due to differences in tape cassette design, a tape recorded by one model of recorder may not be readable by another model of recorder.*

For exact operation details and specific cautions you should observe with your cassette recorder, refer to the owner's manual that accompanied the machine.

Other Accessories

Texas Instruments is engaged in several exciting programs to provide additional accessories and peripherals for your Home Computer. Keep in touch with your dealer to learn details about these products as they become available.

USING THE BUILT-IN PROGRAMS

Now that you have some familiarity with the computer, let's look at the built-in features – TI BASIC and the Equation Calculator.

TI BASIC

Learning to program TI BASIC on your Home Computer can add a useful and enjoyable dimension to many aspects of your life. You can develop programs that range from music and color graphic creations, to games, to personal and business records, to complex mathematical and statistical applications.

TI BASIC is a powerful computer language, yet it's easy to learn and apply. Whatever your level of programming experience, we can help you begin programming quickly.

For those who have no programming experience, we have included the book *Beginner's BASIC*. This book is designed to lead you into programming. It contains many examples and lets you learn TI BASIC through an enjoyable "hands-on" approach.

If you have some programming experience, but have never programmed with BASIC, you may want to consult an excellent intermediate level book by Herbert D. Peckham. This TI/ McGraw-Hill book, *Programming BASIC with the TI Home Computer*, can be ordered from Texas Instruments using the coupon included in *Beginner's BASIC* and should also be available locally.

If you have programmed with BASIC on other computers, you will probably know most of the information in *Beginner's BASIC*. You may want to scan through the book briefly to acquaint yourself with TI BASIC. Then proceed to the "BASIC Reference" section of this book which begins on page 30. This section provides you with a detailed description of TI BASIC. A series of applications programs are included at the end of this section to help you gain a good working knowledge of the computer.

If you have extensive programming experience, you can immediately start programming on the Home Computer. TI BASIC is compatible with the American National Standard for Minimal BASIC.

Equation Calculator

The second built-in program of the Home Computer is an Equation Calculator. This unique calculator is available for a wide variety of applications. The next section of this book explains exactly how to use the calculator for both simple and more advanced calculations.

If you're ready, let's move ahead and get some hands-on experience using both the Equation Calculator and TI BASIC.

Equation Calculator

The Equation Calculator lets you take advantage of the computing power of the Home Computer without entering formal programs. The versatility and convenience of this feature can be applied to everyday arithmetic problems as well as to advanced mathematical operations.

In addition to the capabilities of a high performance calculator, the Equation Calculator has a unique "visible memory" display that shows you the computation in progress. The data stored in memory, the equation being solved, and the keystrokes you enter are displayed in separate sections of the screen. You can easily see what has been done and change values whenever necessary.

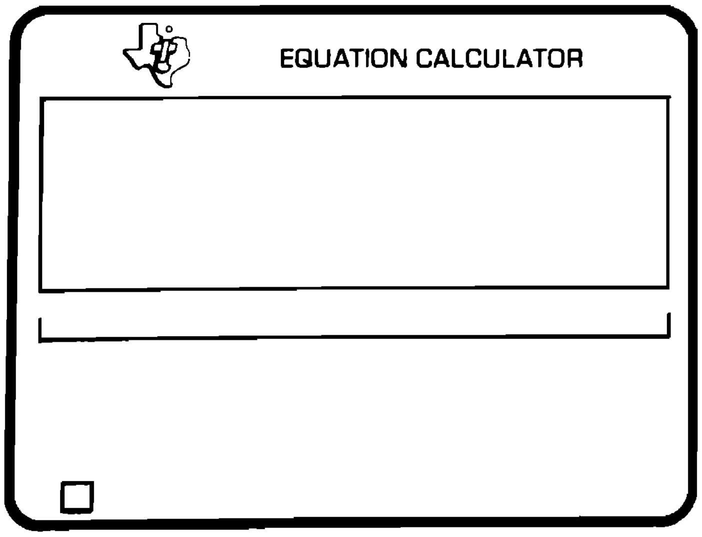
The best way to learn about the capabilities of your Equation Calculator and how to use it is to turn on the computer, select EQUATION CALCULATOR, and work along as you read through this section of the book.

SELECTING THE EQUATION CALCULATOR

When you turn on your Home Computer, you'll see the master title screen. Press any key, and the master selection list is displayed on your screen. Press the **2** key to select EQUATION CALCULATOR.

DISPLAY FORMAT

After you press the **2** key the following screen appears:



Notice that the EQUATION CALCULATOR screen is divided into three special-purpose areas. Let's look at these areas, starting at the bottom of the screen. The section on the lower part of the screen is your "work" area. When you "type" in data, it appears first in this section. When you are working on simple numeric calculations, the answers are displayed in this area.

The middle section of the screen is the "equation memory" area. This area will be blank or will display the equation or formula you are currently solving.

The "variable memory" box at the top of the screen displays up to ten variables and the current value you have assigned to each variable. When you are solving a formula or equation to find the value of a variable, the value (or solution) will be labelled and displayed in this area of the screen. When this area is filled (with ten variables and their values), the next variable and its value entered will be displayed in the "work area" at the bottom of the screen. (Variables and their use are discussed in detail in "More Advanced Calculations" on page 23.)

As we go along and actually work through some examples you'll see exactly how each section of the display functions. But before we try some problems, review "A Tour of the Keyboard" on pages 10-12 of this book.

SPECIAL KEY FUNCTIONS

ENTER key

The **ENTER** key is used to enter variables and to complete a calculation. For example, to add 2 plus 5:

press **2**
then press **SHIFT +**
then press **5**
then press **ENTER**

After you press **ENTER**, the results are displayed on the screen.

SHIFT = (Equals Key)

The **SHIFT =** key is used only in assigning a value to a variable. Examples:

INTEREST=650
X=Y+6
VOL=25
A=3

SHIFT ↑ (UP)

The up-arrow key is used to move an equation, formula, or other expression from the bottom line of the "work" area of the screen into the "equation memory" area of the screen. If an expression is already showing in the "equation memory" area, pressing **SHIFT** ↑ replaces that expression with whatever expression is on the bottom line of the work area. If the bottom line is blank, pressing **SHIFT** ↑ clears the "equation memory" area. You must press this key *before* you press **ENTER.**

SHIFT ↓ (DOWN)

Pressing the down-arrow key brings the expression in the equation memory area to the bottom line of the work area. (It also remains displayed – and stored – in the equation memory area.) You'll want to bring an expression down from the memory area when you are ready to solve (or execute) it. Also you can bring an expression back to the work area to edit – or change – it in some way. (We'll discuss this in more detail later.) You must press this key *before* you press **ENTER**.

SHIFT → (RIGHT)

The right-arrow (forwardspace) key moves the cursor (□) to the right without erasing characters as it passes across them.

SHIFT ← (LEFT)

The left-arrow (backspace) key is used to move the cursor (□) to the left without erasing the characters it passes over.

SHIFT F (DEL)

The delete key is used to delete a number, letter, or other character you've typed on the bottom line of the work area (before you press **ENTER**). Using **SHIFT** ←, backspace to the character you want to delete. Press **SHIFT F**. The character is deleted, and any numbers or letters following the deleted character on the line are automatically moved one space to the left.

SHIFT G (INS)

The insert key is used to insert a letter, number, or other character into the information you have typed on the bottom line of the work area (before you press **ENTER**). Backspace (using **SHIFT** ←) to the point where you want to insert a new character. Press **SHIFT G**. Then type in the new character. The new character is inserted and all the other characters on the line are moved one space to the right. Any characters that are shifted off the end of the line are lost.

SHIFT T (ERASE) or **SHIFT C** (CLEAR)

If you want to clear the bottom line of the work area (before you press **ENTER**), press either **SHIFT T** or **SHIFT C**.

USING THE CALCULATOR

Simple Calculations

Let's try working a few problems so you can see exactly how the calculator operates. We'll begin with some simple calculations. First let's calculate the problem 5+3+4 and see how it appears on the display.

- First, press **5**
- then press **SHIFT +**
- then press **3**
- then press **SHIFT +**
- then press **4**
- then press **ENTER**

(Note that you press **ENTER,** *not* **SHIFT =**.)

Notice that the "variable memory" box and the "equation memory" line are empty. The lower "work" area looks like this:

```
5+3+4
  12
□
```

The answer (12) is printed on the line below your problem entry. When you press **ENTER** the bottom line "scrolls" up one line. After the calculator prints the answer, the bottom line again "scrolls" up one line. The work area displays up to six (6) lines of input at one time. When the seventh line is entered at the bottom, the top line in the work area scrolls off the screen.

You can type in a calculation or expression that contains up to 28 characters (one full line). When you have typed the maximum characters allowed on the line, you'll hear a "beep." If you press **ENTER** at this point, the entries on the line are calculated, and the answer is displayed on the next line. If you have a long calculation – such as adding up all of your checks for the month – you'll want to divide the entries and make a series of calculations. Try entering several addition or subtraction problems and see how the lines scroll up on the screen.

POSITIVE AND NEGATIVE NUMBERS

You can enter either positive or negative numbers in your calculations. For positive numbers, the plus sign (+) is assumed. You do not have to type it in, and the Equation Calculator does not print it on the screen. For negative numbers, type a minus sign (–) in front of the number. The calculator also prints a minus sign before the number for negative numbers.

ORDER OF OPERATIONS

Thus far we've only experimented with addition and subtraction problems. At times, however, you'll want to solve problems that have more than one operation involved. Consider this problem:

4+10–6/2*3

You can get several different answers to this problem according to the order in which the operations are done. For example,

4+10–6/2*3		4+10–6/2*3
=14–6/2*3	or	=4+10–3*3
=8/2*3		=4+10–9
=4*3		=14–9
=12		=5

Which is correct? Type 4 +10–6/2*3, press **ENTER**, and see what answer the Equation Calculator gives you. Is the answer "5"? This is the correct answer. Mathematics has a requirement that there be only one correct solution for any computation. To assure this, there is a commonly accepted order in which arithmetic operations are performed. Your computer performs calculations in this order. In any problems involving mixed calculations – addition, subtraction, multiplication, and division – the arithmetic operations are completed in the following order:

1. Multiplication and division are performed first.
2. Then addition and subtraction are performed.

With the Equation Calculator you can enter a problem directly, from left to right, and the computer automatically sorts the numbers and operations and computes them according to the above rules.

At times you may want to specify the exact order in which an expression is evaluated. In these cases, you use the parentheses ( ) keys to group numbers and operations so that the problem is solved in the order you indicate. The computer completes the computations inside the parentheses first. So the new order of operations becomes:

1. Operations inside the parentheses
2. Multiplication and division operations
3. Addition and subtraction operations

Let's look at how the position of the parentheses alters the answer you get in a problem. Try the following problem:

58+10/2*32

If we enter the problem just like this, we get an answer of 218 because:

58+10/2*32
=58+5*32
=58+160
=218

Equation Calculator

By adding parentheses in different places, we get a variety of answers. Try (58+10)/2*32:

(58+10)/2*32
=68/2*32
=34*32
=1088

Or try (58+10/2)*32:

=(58+5)*32
=63*32
=2016

Experiment with some problems of your own. Notice the difference the parentheses make in computing your problems. (Note: Sometimes you'll see parentheses used to *imply* multiplication, such as (2+1)(3+2)=15. *Your computer will not perform implied multiplications.* You must type the multiplication symbol (*) between the parentheses.)

More Advanced Calculations

The Equation Calculator is useful for simple arithmetic calculations. But it can do so much more! In this section we'll look at some of the other ways you can use your calculator.

At times you may want to make several similar calculations in which you change only one or two of the numbers. So, instead of typing in the entire problem each time, you can give a special English-like name to the number you want to change. Then you only have to assign a value (or number) to the special name. This name is called a "variable." In other words, its value can "vary" from problem to problem.

Let's consider an example. You are shopping for furniture. You have located a sofa and a table that you like, but are trying to decide between two different chairs. So you decide to compare the total cost in both cases. The costs are:

sofa:	$575
table:	$125
chair:	$305
chair:	$267

So. TOTAL=575+125+CHAIR

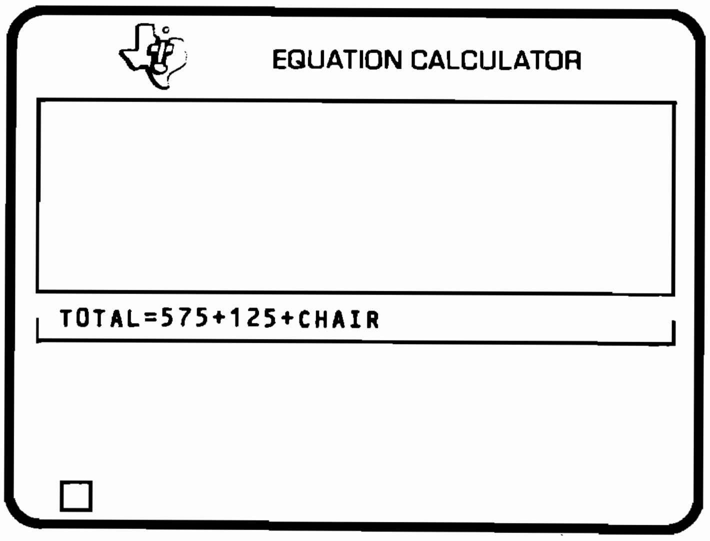
"CHAIR" and "TOTAL" are the names we've given to the *variables* in this problem. The entire expression is called an *equation.* Let's see how the Equation Calculator handles this problem.

First, type the word NEW and press **ENTER**. This clears the screen and the memory areas. Then type in the equation:

TOTAL=575+125+CHAIR

(DO NOT PRESS **ENTER** WHEN YOU FINISH TYPING AN EQUATION.)

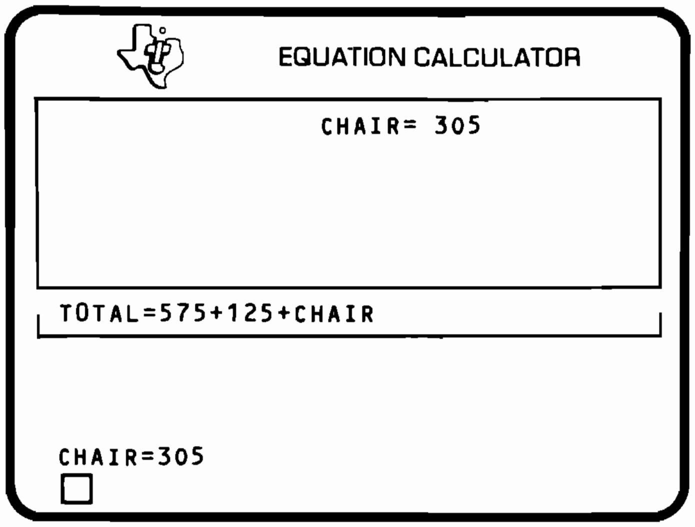
The equation appears on the bottom line of the work area. Let's store the equation in the "equation memory" area since we'll use it more than once. To move it to the memory area, press **SHIFT** ↑. Immediately, the equation moves from the bottom line of the work area to the "equation memory" line. This is how your screen looks:



```
TOTAL=575+125+CHAIR

□
```

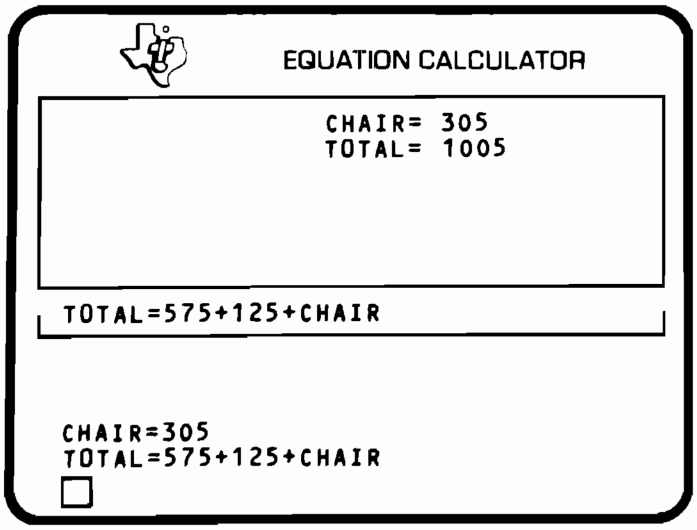
Equation Calculator

Now let's assign the first value to CHAIR. Type CHAIR=305. You'll see CHAIR=305 on the bottom line of the work area. Press **ENTER**. When you press **ENTER**, CHAIR=305 moves up one line in the work area, and it is also stored in the "variable memory" box. Your screen now shows:



```
EQUATION CALCULATOR

                CHAIR= 305

TOTAL=575+125+CHAIR

CHAIR=305
□
```

To calculate TOTAL when CHAIR=305, press **SHIFT** ↓ (to bring the equation back to the work area), and press **ENTER**. You now see the answer, TOTAL=1005, in the "variable memory" box:



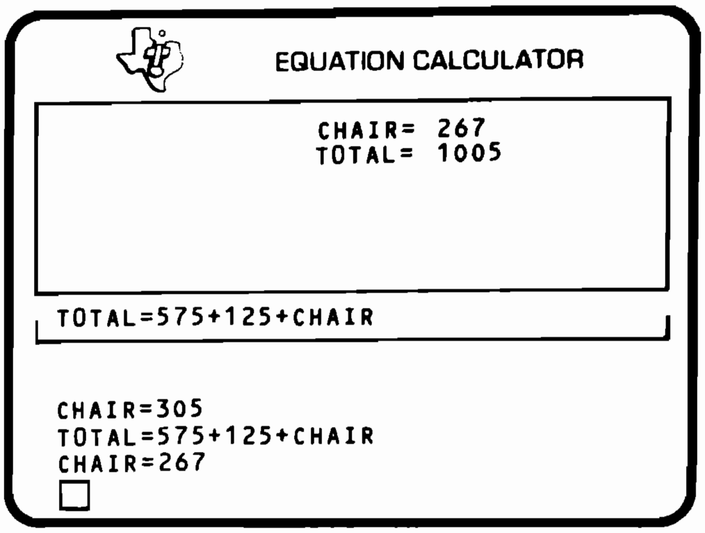
```
EQUATION CALCULATOR

                CHAIR= 305
                TOTAL= 1005

TOTAL=575+125+CHAIR

CHAIR=305
TOTAL=575+125+CHAIR
□
```

To find TOTAL when CHAIR=267, first type

CHAIR=267

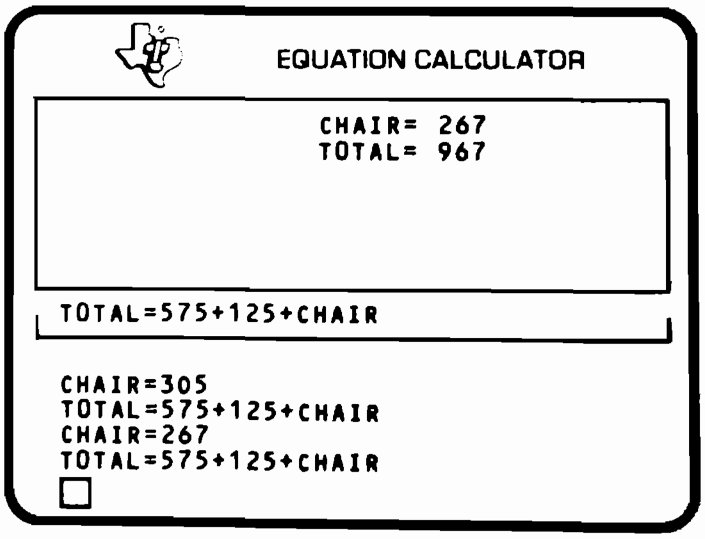
Then, press **ENTER**.

The value shown for CHAIR in the "variable memory" box changes to 267 as shown:



```
EQUATION CALCULATOR

                CHAIR= 267
                TOTAL= 1005

TOTAL=575+125+CHAIR

CHAIR=305
TOTAL=575+125+CHAIR
CHAIR=267
□
```

Next, bring the equation to the work area again by pressing **SHIFT** ↓, and press **ENTER** to tell the Equation Calculator to evaluate it. When you press **ENTER**, the computer shows the new value for TOTAL (967) in the "variable memory" box at the top of the screen:



```
EQUATION CALCULATOR

                CHAIR= 267
                TOTAL= 967

TOTAL=575+125+CHAIR

CHAIR=305
TOTAL=575+125+CHAIR
CHAIR=267
TOTAL=575+125+CHAIR
□
```

You can use almost any letter, group of letters, or word for a variable name. Several abbreviations and words *cannot* be used as variable names because they are reserved for special functions in TI BASIC. (You can use them as a part of a variable name, however.)

Equation Calculator

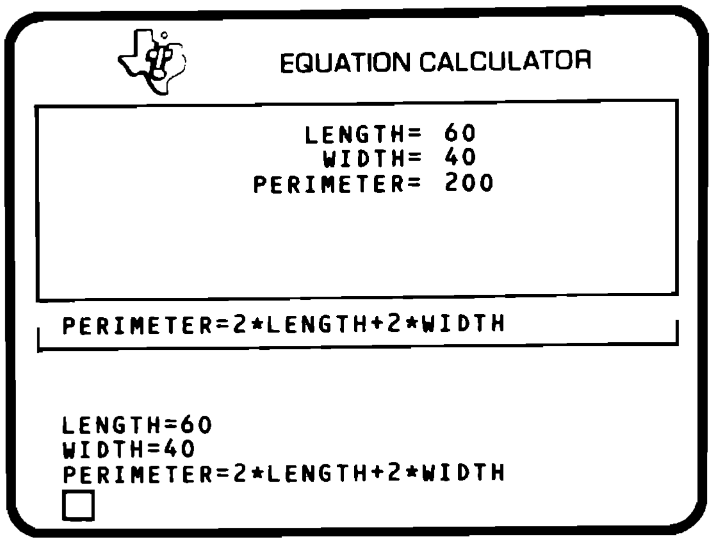
You'll find a list of these words on page 44 of the "BASIC Reference" section of this book. The variable name can be up to fifteen characters in length. You cannot leave spaces between characters in the name, and the only characters allowed are letters, numbers, the "at" sign (@), and the line key (__). If you try to enter a variable name that's too long, has spaces between the characters, or is a reserved word, etc., you'll hear a tone and see an error message such as "BAD NAME" or "INCORRECT STATEMENT." (See page 168 for a complete list of error messages.) If you do get an error message, just type in the name again, correctly.

Let's try another example. You want to fence in a part of your yard and want to find the distance around this rectangular area (or the "perimeter"). You may remember the formula PERIMETER=2*LENGTH + 2*WIDTH.

First, type NEW and press **ENTER** to clear all other entries. Then type in the formula. Let the LENGTH=60 and the WIDTH=40. Here's what you do:

Procedure	*Comments*
1. Type PERIMETER=2* LENGTH+ 2*WIDTH	This is the formula used to find PERIMETER
2. Press **SHIFT** ↑	To store the formula in the "equation memory" area
3. Type LENGTH=60	To give the variable (LENGTH) a value
4. Press **ENTER**	To enter LENGTH=60
5. Type WIDTH=40	To give the variable (WIDTH) a value
6. Press **ENTER**	To enter WIDTH=40
7. Press **SHIFT** ↓	To bring the formula (or equation) back to the work area
8. Press **ENTER**	To tell the Equation Calculator to find the answer

The answer, PERIMETER=200, is displayed in the variable memory box at the top of the screen. The entire screen looks like this:

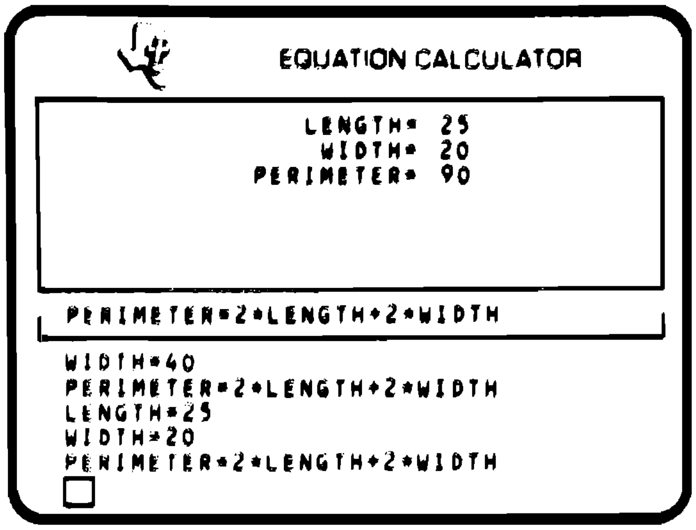


```
        EQUATION CALCULATOR

            LENGTH= 60
             WIDTH= 40
         PERIMETER= 200

 PERIMETER=2*LENGTH+2*WIDTH

 LENGTH=60
 WIDTH=40
 PERIMETER=2*LENGTH+2*WIDTH
 □
```

If you want to change the value assigned to one or both of the variables LENGTH and WIDTH, simply type in the new values and press **ENTER**. Then press **SHIFT** ↓, and press **ENTER** to get the new value for the variable PERIMETER. Let's look at this procedure once again, letting LENGTH=25 and WIDTH=20.

Procedure	*Comments*
1. Type LENGTH=25 and press **ENTER**	"25" replaces "60" in the variable memory box
2. Type WIDTH=20 and press **ENTER**	"20" replaces "40" in the variable memory box
3. Press **SHIFT** ↓	To bring formula to work area
4. Press **ENTER**	To tell the calculator to compute PERIMETER using the new values

This is how the screen looks now:



```
                EQUATION CALCULATOR

                      LENGTH=  25
                       WIDTH=  20
                   PERIMETER=  90

 PERIMETER=2*LENGTH+2*WIDTH

 WIDTH=40
 PERIMETER=2*LENGTH+2*WIDTH
 LENGTH=25
 WIDTH=20
 PERIMETER=2*LENGTH+2*WIDTH
 □
```

(Notice that the original first line (LENGTH=60) scrolls off the top of the work area when you enter the seventh line.)

If you have a long equation, you may need to use an abbreviation or just a single letter as the variable name so that it will fit on one line. We could have written the equation above (PERIMETER=2*LENGTH+2*WIDTH) as P=2*L+2*W.

You can recall the value of any variable, whether or not it is still displayed on the screen, by typing the variable name and pressing **ENTER**.

Anytime you ask the calculator to evaluate a variable that is still undefined (by pressing **ENTER**), the computer will display the value as zero (0). For example, if you type X=A+B and press **ENTER**, the calculator will display the value X=0 in the variable memory box because you have not assigned a value to the variables, A and B. For the same reason, if you type in the variable INTEREST (without assigning it a value) and press **ENTER**, the calculator displays INTEREST=0 in the work area at the bottom of the screen.

Try entering the following problems, and see how the computer displays your entries and the final answer.

1. Find the Miles Per Gallon (MPG). You have driven 350 miles and put 12.5 gallons of gas in your car. The equation for this problem is:
   MPG=MILES/GALLONS

   (Ans: 28)

2. (a) Find the annual rate of interest on a $250 Treasury Bill that is sold for $244.33 and matures in 90 days. The equation for this problem is:
   R=(I/(P*T))*100 (to get the answer in %)
   where
   R=annual rate of interest
   I=interest earned in dollars
   ($250.00−$244.33)
   P=principal ($244.33)
   T=time in years (90/365)
   (*HINT*: The answer should be 9.411451725 or 9.41%.)

   (b) Solve the same problem as 2(a), but change the maturity time to 60 days.

   (Ans: 14.12%)

Now that you have solved some equations with your Equation Calculator, let's explore some of the additional capabilities of this Home Computer feature.

Built-In Functions

The Equation Calculator can take advantage of many of the special functions of TI BASIC. These special functions include such areas as:

- expressing very large or very small numbers in scientific notation
- exponents and roots
- absolute values
- trigonometric functions
- logarithms and antilogarithms
- string functions
- and others

Equation Calculator

For complete information about all of the functions, see the "BASIC Reference" section of this book or pages 127-141 of *Beginner's BASIC*. You do not have to follow all of the programming conventions, however, when you are working with the Equation Calculator. You *do not* have to type "PRINT" or "LET." Let's look at exactly how you enter some of these operations in the calculator.

Scientific Notation

To enter 98765432100 in scientific notation, type:

```
9.87654E+10
```

To enter 0.000000000000123 in scientific notation, type:

```
1.23E-13
```

Exponents

Quite often in mathematical calculations, we must raise some number to a power. The caret or *exponentiation symbol* (∧) tells the computer that the number following it is a power.

To enter 8^3, type

```
8∧3
```

To enter 25^2, type

```
25∧2
```

To enter $y=x^3$, type

```
Y=X∧3
```

Roots

Since many calculations call for finding the *square root* of a number, this function is built into the computer. The letters SQR stand for "square root of" and instruct the computer to find the square root of the number, variable, or expression within the parentheses following the letters.

To enter "A =the square root of 4", type:

```
A=SQR(4)
```

Other roots must be computed by using a form of exponentiation. Computing a root of a number is the same function as raising the number to a power which is the *reciprocal* of the root. Therefore, $\sqrt[3]{125}$ is the same as $125^{(1/3)}$

To enter $\sqrt[3]{125}$, type:

```
125∧(1/3)
```

Note that we must use parentheses around the exponent (1/3).

Trigonometric Functions

The following trigonometric functions are available in your calculator:

SIN( )—Finds the *sine* of the number or numeric expression enclosed in parentheses.

COS( )—Finds the *cosine* of the number or numeric expression enclosed in parentheses.

TAN( )—Finds the *tangent* of the number or numeric expression enclosed in parentheses.

ATN( )—Finds the *arctangent* of the number or numeric expression enclosed in parentheses.

All trigonometric functions are performed by the calculator in radians, rather than degrees. Therefore, if your data is measured in degrees, you'll need to convert the measurement to radians before using it with the function. To convert an angle from degrees to radians, multiply by $\pi/180$. To convert from radians to degrees, multiply by $180/\pi$.

To enter the *sine* of X, where X is the angle expressed in radians, type:

```
SIN(X)
```

Other trigonometric functions are entered in the same manner, using the correct abbreviation.

When using any of these special TI BASIC functions with the calculator, you follow the same procedures as you do with simple calculations or equations. In other words, you continue to press **ENTER**, or **SHIFT** [↑], or **SHIFT** [↓] as described previously.

Equation Calculator

Calculation Overflow or Underflow

The Equation Calculator feature of your computer has the same numeric range as TI BASIC. When an overflow occurs, a warning is given with the message "NUMBER TOO BIG." When an underflow occurs, the computer replaces the value of the number with a zero. No warning or error message is given. (See page 37 of the "BASIC Reference" section of this book for a complete discussion of the numeric range of the computer.)

Editing

Editing the information you have typed into the calculator or changing (or correcting) the equation in the "equation memory" area is very easy. If you want to make a change in the line you have just typed (before you press **ENTER**) simply backspace, using **SHIFT** [←], to the point of change. Then

Press **SHIFT F** to delete a character
Press **SHIFT G** to insert a character

Then forwardspace, using **SHIFT** [→], to return to the place you were before backspacing.

To correct errors simply type over the incorrect entries. You can erase characters by pressing the **SPACE BAR**.

If you want to change or correct the equation in the "equation memory" area, press **SHIFT** [↓] to bring the equation to the bottom line of the work area. Next, correct or change the equation as explained above. Then press **SHIFT** [↑] to place the corrected expression in the "equation memory" line.

Print Separators

If you desire, you can tell the calculator where to print the answers that appear in the work area by using commas, semicolons, colons, or the TAB function. See "Print Separators" and "Tab Function" on pages 95-96 of the "BASIC Reference" section of this book or pages 57-66 of *Beginner's BASIC* for a complete discussion of possible print formats.

Special Features

Key Words

There are several words you type to give the Equation Calculator special instructions. These words are:

NEW – You type NEW to tell the computer to clear all previous entries. This clears the entire screen and all calculator memories.

BYE – You type BYE to tell the computer to leave the Equation Calculator and return to the master title screen.

LET – You may (but need *not*) type LET when assigning a value to a variable (Example, LET A=35).

PRINT – You may (but need *not*) type PRINT to tell the computer to print an expression (Example, PRINT A*2).

(*Note:* All of these are TI BASIC commands or statements, and you must press **ENTER** after typing them.)

Error Messages and Tones

The computer gives several messages in case of errors. See the "BASIC Reference" section of this book for a complete listing of these messages and their meanings.

At times you'll hear warning tones when you are working with your calculator. The most important tone you'll hear is the one indicating you have typed the last allowed character on the bottom line of the work area.

The many features of the Equation Calculator make it a powerful tool that you can use for many applications. In addition to its usefulness as a calculator, it can also serve as an introduction to programming. As you've discovered, the calculator uses most of the functions of TI BASIC. The next section of this book and the separate book *Beginner's BASIC* help you move from the powers of the Equation Calculator into true programming in TI BASIC.

BASIC Reference Section

BASIC Reference Section

Introduction

This section of your User's Reference Guide provides a complete explanation of all of the commands and statements that are a part of the TI BASIC language built right into your home computer. As mentioned earlier, BASIC is a computer language designed to be easy for beginners to use, yet powerful enough to allow you to use your computer for a whole host of applications. There are three different paths available to help you learn TI BASIC.

If you're a beginner – and have never had any experience with programming – the best place to begin is with the *Beginner's BASIC* book included with your home computer. The book is intended to be an enjoyable, quick, self-paced first experience with programming in TI BASIC. Once you've become familiar with BASIC, this reference guide will provide the in-depth, ready reference to terms and information you'll want at your fingertips as you enjoy the experience of programming.

If you've had some programming experience – and just want to get familiar with TI BASIC and how it works on your home computer – we've provided a series of applications programs at the end of this manual. These programs start out at a very simple level and progressively become more complex. Exploring these programs will illustrate for you the use of many of the statements in TI BASIC. This reference manual provides in-depth information when you need it.

For those of you with some programming experience who may not have programmed in BASIC or who want to "brush up" as you begin using your home computer, we recommend that you begin with Herbert Peckham's excellent book, *Programming BASIC with the TI Home Computer,* which provides a rapid, higher-level learning experience in BASIC. It is available at most popular bookstores.

For the knowledgeable – once you've gained proficiency in programming – this guide will serve as your primary reference on TI BASIC statements and commands, providing those details that need refreshing from time to time. TI Home Computer BASIC conforms to the American National Standard for Minimal BASIC. Additional features in TI BASIC such as color graphics, sound, and many others are also described in this manual. If you are an experienced BASIC programmer, you should have little trouble jumping right into TI BASIC and using it.

Basic Reference Section

How This Section Is Organized

This reference guide is organized with usability as the key goal, and is divided into the following functional groups.

1. General Information
2. Commands
3. General Program Statements
4. Input-Output Statements
5. Color Graphics and Sound
6. Built-In Numeric Functions
7. Built-In String Functions
8. User-Defined Functions
9. Arrays
10. Subroutines
11. File Processing

A glossary of often-used terms is found in the back of this manual.

Notational Conventions

At the beginning of the discussion for each TI BASIC command or statement, a line appears which shows the general format for entering the command or statement. Certain notational conventions have been used in these format lines. These conventions are discussed here to help you understand how they are used.

{ } – The braces indicate that you have a choice of what to use. You may use only one of the items given within the braces.

[]– The brackets indicate that the item within is optional. You may use it if you wish, but it is not required.

. . . – The ellipsis indicates that the preceding item may be repeated as many times as you desire.

CAPITAL LETTERS – Words appearing in all capital letters must be typed exactly as shown if you choose to use that item.

italics – Words appearing in italics are a general description of the item or items that need to appear there. When words are printed in italics, you need to enter your own choice in place of the italicized words when you enter the statement or command.

Examples

For each statement or command in this manual, program examples are shown at the right. Each line you must enter is indicated by the prompt character (>) to the left of the line, just as it would appear on the screen. For statements which will take up more than one line on the screen, the prompt character appears only at the beginning of the statement. Anything which the computer places on the screen does not show the prompt character. If a new program is needed, the NEW command is shown in the example. It is best if you enter the examples into the computer and try them as you read the explanation. Of course, you are encouraged to enter your own examples as well.

General Information

Introduction

Once your computer is set up, it is a simple process to begin using TI BASIC. When you turn on your computer, the master computer title screen appears. Press any key on the keyboard to get the master selection list to be displayed. When the master selection list appears, press the **1** key to select TI BASIC. The screen is now blank except for the words "TI BASIC READY" and a prompt character (>) followed by a flashing cursor (□). Whenever the cursor is on the screen, the computer is waiting for you to enter something. The prompt character marks the beginning of each line you type.

Each line of the screen can display up to 28 characters. Each statement or command may be up to four screen lines in length. When you have completely filled one screen line, the cursor automatically moves down to the next line as you continue typing. When you have completely filled four lines, the computer will accept more characters, but the cursor will remain in the same position. Each character you enter will replace the last character of the line.

All of the keys discussed in the Special Keys section may be used in editing program lines before you press the **ENTER** key. To change anything in a program line after you have pressed **ENTER**, you can retype the entire program line making the desired corrections as you type in the line again or you can enter Edit Mode. For information on using Edit Mode, see page 66. Note that whenever you do any editing on a program, all open files are closed (see OPEN statement, page 145), and all variables become undefined.

The remainder of this section gives information which applies to many commands and statements in TI BASIC.

Examples:

```
 TI BASIC READY

>□
```

```
>NEW

>10 A=2
>RUN

  ** DONE **

>PRINT A
  2

>20 B=3
>PRINT A
  0
```

Special Keys

Several keys have special functions in TI BASIC. These keys are discussed here.

ENTER – When you press the **ENTER** key, the computer accepts the program line you have just finished typing. Remember that you may use up to four screen lines for each program line before you press **ENTER**.

SHIFT Q (QUIT) – When you press the Quit key, the computer leaves TI BASIC and returns to the master computer title screen. When the computer leaves TI BASIC, the program and all data stored in memory is erased. Note that this key does not close open files (see OPEN statement, page 145). Thus, it is preferable to use the BYE command (see page 52) to leave BASIC.

SHIFT ↑ (UP) – The Up-Arrow key works exactly like the **ENTER** key, except in Edit Mode (see page 66).

SHIFT ↓ (DOWN) – The Down-Arrow key works exactly like the **ENTER** key, except in Edit Mode (see page 66).

SHIFT ← (LEFT) – The Left-Arrow (backspace) key moves the cursor one position to the left every time it is pressed. When the cursor moves over a character it does not delete or change it in any way. If the cursor reaches the beginning of the line, pressing the Left-Arrow key has no effect.

SHIFT → (RIGHT) – The Right-Arrow (forwardspace) key moves the cursor one position to the right each time it is pressed. Using this key allows you to move the cursor over a character without deleting or changing it in any way. If the cursor reaches the end of the line (4 screen lines), pressing the Right-Arrow key has no effect.

SHIFT G (INS) – The Insert key is used to insert characters in the middle of a program line. To insert characters, position the cursor (using **SHIFT** ← or **SHIFT** →) over the character immediately to the right of the place where you wish to insert characters, then press the Insert key. After you have pressed the Insert key, each time you press a character, the cursor and every character of the program line that is not to the left of the cursor is moved one position to the right. The character corresponding to the key you pressed is then inserted in the blank position left by the shifting of the cursor and other characters. Note that characters shifted off the end of the program line are deleted from the line. When you have finished inserting characters, press any other special key listed above, except Quit (**SHIFT Q**).

Special Keys

SHIFT F (DEL) – The Delete key is used to delete characters from the program line. To delete characters, position the cursor (using **SHIFT** ← or **SHIFT** →) over the character you wish to delete, then press the Delete key. When you press the Delete key, the character under the cursor is deleted and all characters of the program line to the right of the cursor are moved one position to the left. The cursor does not move. A blank space is used to fill the position at the right end of the program line left by the shifting of the characters.

SHIFT C (CLEAR) – the Clear or Break key has *two* functions, depending on when you use it.

- When this key is pressed while a program is running, a breakpoint (see page 58) will be taken at the next program line to be executed. This key allows you to temporarily stop a program while it is running. Note that you must continue to hold the Break key until the program stops running. When you stop running a program using the Break key, the message "BREAKPOINT AT line-number" is displayed. The program line designated by the line-number has not been performed. You can start the program running again where you stopped by entering the CONTINUE command (see page 63).
- When the Clear key is pressed while typing in a program line, the line scrolls up on the screen and is not entered. This key has additional functions in Edit Mode (see page 67) and in Number Mode (see page 55).

SHIFT T (ERASE) – the Erase key erases the entire program line which you are typing. The line is not entered. This key works differently in Edit Mode (see page 67) and Number Mode (see page 55).

SPACE BAR – the space bar moves the cursor one position to the right each time it is pressed. If you move the cursor over a character using the space bar, that character is replaced by the space character.

SPACE – the **SPACE** key works just like the SPACE BAR.

Blank Spaces

In general, a blank space can occur almost anywhere in a program without affecting the execution of the program. However, any extra blank spaces you put in that are not required will be deleted when the program line is displayed by the EDIT (see page 66), NUM (see page 53), or LIST (see page 49) command. There are some places where blank spaces must not appear, specifically:

(1) within a line number (see page 36)
(2) within a reserved word (see page 44)
(3) within a numeric constant (see page 37)
(4) within a variable name (see page 39)

The following are some examples of incorrect use of blank spaces. The correct line appears in the column at the right.

(1) 1 00 PRINT "HELLO"
(2) 110 PR INT "HOW ARE YOU?"
(3) 120 LET A = 1 00
(4) 130 LET CO ST = 24.95

All reserved words (see page 44) in a program should be immediately preceded by and followed by one of the following:

- a blank space
- an arithmetic operator (+ − */ ∧)
- the string operator (&)
- a special character used in a particular statement format (< = >(),;:#)
- end of line (**ENTER** key)

Examples:

```
>100 PRINT "HELLO"
>110 PRINT "HOW ARE YOU?"
>120 LET A=100
>130 LET COST=24.95
```

Line Numbers

Each program is comprised of a sequence of BASIC language program lines ordered by line number. The line number serves as a label for the program line. Each line in the program begins with a line number which must be an integer between 1 and 32767, inclusive. Leading zeroes may be used but are ignored by the computer. For example: 033 and 33 will be read as 33. You need not enter lines in sequential order; they will be automatically placed that way by the computer.

When you run the program, the program lines are performed in ascending sequential order until:
(1) a branch instruction is performed (see General Program Statements, page 72)
(2) an error occurs which causes the program to stop running (see page 169)
(3) the user interrupts the running of the program with a BREAK command (see page 58) or by using the Break key (**SHIFT C**)
(4) a STOP statement (see page 76) or END statement (see page 75) is performed
(5) the statement with the largest line number is performed

If you enter a program line with a line number less than 1 or greater than 32767, the message "BAD LINE NUMBER" will be displayed and the line will not be entered into memory.

Examples:

```
>NEW

>100 A=27.9
>110 B=31.8
>120 PRINT A;B
>130 END


>RUN
  27.9  31.8

 ** DONE **
```

```
>0 A=2

 * BAD LINE NUMBER

>33000 C=4

 * BAD LINE NUMBER
```

Numeric Constants

Numeric constants must be either positive or negative real numbers. You may enter numeric constants with any number of digits. Values are maintained internally in seven radix-100 digits. This means that numbers will have 13 or 14 decimal digits depending on the value of the number.

Examples:

```
>PRINT 1.2
  1.2

>PRINT -3
 -3

>PRINT 0
  0
```

Scientific Notation

Very large or very small numbers are easily handled using scientific notation. A number in scientific notation is expressed as a base number (mantissa) times ten raised to some power (exponent).

$$\text{Number} = \text{Mantissa} \times 10^{\text{Exponent}}$$

To enter a number using scientific notation:

First, enter the mantissa (be sure to enter a minus sign first if it's negative).

Enter the letter "E."

Enter the power of 10 (if it is negative, enter the minus sign before you enter the exponent).

The following are some examples of how numbers in scientific notation are entered.

Number	*Entered as*
3.264×10^4	3.264E4
-98.77×10^{21}	−98.77E21 or −9.877E22
5.691×10^{-5}	5.691E−5
-2.47×10^{-17}	−2.47E−17

```
>PRINT 3.264E4
  32640

>PRINT -98.77E21
 -9.877E+22
```

Numeric constants are defined in the range of −9.9999999999999E127 to −1E-128, 0, and 1E-128 to 9.9999999999999E127.

```
>PRINT 0
  0
```

Underflow – If an entered or computed number, when rounded, is greater than −1E-128 and less than 1E-128, then an underflow occurs. When an underflow occurs, the computer replaces the value of the number with a zero and the program continues running. No warning or error is given.

```
>PRINT -9E-130
  0

>PRINT 9E-142
  0
```

Overflow – If a number is entered or computed whose value when rounded is greater than 9.9999999999999E127 or less than −9.9999999999999E127, an overflow occurs. When an overflow occurs, the constant is replaced by the computer's limit, a warning is given with the message "NUMBER TOO BIG," and the program continues running. The computer's limit is −9.9999999999999E127 or 9.9999999999999E127 as appropriate. Note that "**" is printed if the exponent is greater than 99.

```
>PRINT 97E136

 * WARNING:
   NUMBER TOO BIG
  9.99999E+**

>PRINT -108E144

 * WARNING:
   NUMBER TOO BIG
 -9.99999E+**
```

String Constants

A string constant is a string of characters (including letters, numbers, spaces, symbols, etc.) enclosed in quotes. Spaces within string constants are not ignored and are counted as characters in the string. All characters on the keyboard that can be displayed may be used in a string constant. A string constant is limited by the length of the input line (112 characters or four lines on the screen).

Examples:

```
>NEW

>100 PRINT "HI!"
>110 PRINT "THIS IS A STRING
 CONSTANT."
>120 PRINT "ALL CHARACTERS (+
 -*/ @,) MAY BE USED."
>130 END
>RUN
 HI!
 THIS IS A STRING CONSTANT.
 ALL CHARACTERS (+-*/ @,) MAY
 BE USED.

 ** DONE **
```

When a PRINT (see page 93) or DISPLAY (see page 98) statement is performed, the surrounding quote marks are not displayed. If you wish to have words or phrases within a string printed with surrounding quote marks, simply enter a pair of adjacent quote marks (double quotes) on either side of the particular word or phrase when you type it in.

```
>NEW

>100 PRINT "TO PRINT ""QUOTE
 MARKS"" YOU MUST USE DOUBLE
 QUOTES."
>110 PRINT
>120 PRINT "TOM SAID, ""HI, M
 ARY!"""
>130 END
>RUN
 TO PRINT "QUOTE MARKS" YOU M
 UST USE DOUBLE QUOTES.

 TOM SAID, "HI, MARY!"

 ** DONE **
```

Variables

In BASIC all variables are given a name. Each variable name may be one or more characters in length but must begin with a letter, an at-sign (@), or the line (_). The only characters allowed in a variable name are letters, numbers, the at-sign (@), and the line (_). One exception is the dollar-sign ($). The last character in a string variable name *must* be a dollar-sign ($) and this is the only place in a variable name that it may be used. Variable names are restricted to fifteen characters including the dollar-sign for string variable names.

Array names follow the same rules as simple variable names. (See the section on Arrays, page 134 for more information.) In a single program, the same name cannot be used both as a simple variable and as an array name, nor can two arrays with different dimensions have the same name. For example, Z and Z(3) cannot both be used as names in the same program, nor can X(3,4) and X(2,1,3). However, there is no relationship between a numeric variable name and a string variable name which agree except for the dollar sign (X and X$ may both be used in the same program).

Numeric Variable Names

Valid: X, A9, ALPHA, BASE_PAY, V(3), T(X,3),
TABLE (X,XX7Y/2)
Invalid: X$, X/8, 3Y

String Variable Names

Valid: S$, YZ2$, NAME$, Q5$(3, X)
Invalid: S$3, X9, 4Z$

If you enter a variable name with more than fifteen characters, the message "BAD NAME" is displayed and the line is not entered into memory. Reserved words (see page 44) are not allowed as variable names, but may be used as part of a variable name. For example, LIST is not allowed as a variable name but LIST$ is accepted.

At any instant while a program is running, every variable has a single value. When a program begins running, the value associated with each numeric variable is set to zero and the value associated with each string variable is set to null (a string with a length of zero characters). When a program is running, values are assigned to variables when LET statements (see page 73), READ statements (see page 89), FOR-TO-STEP statements (see page 81), or INPUT statements (see page 86) are performed. The length of the character string value associated with a string variable may vary while a program is running from a length of zero to a limit of 255 characters.

Examples:

```
>110 ABCDEFGHIJKLMNOPQ=3

 * BAD NAME
```

Numeric Expressions

Numeric expressions are constructed from numeric variables, numeric constants, and function references using arithmetic operators (+ − */∧). All functions referenced in an expression must be either functions supplied in TI BASIC (see sections on Built-In Functions) or defined by a DEF statement (see page 131). The two kinds of arithmetic operators (prefix and infix) are discussed below.

The *prefix* arithmetic operators are plus (+) and minus (−) and are used to indicate the sign (positive or negative) of constants and variables. The plus sign indicates the number following the prefix operator (+) should be multiplied by +1, and the minus sign indicates the number following the prefix operator (−) should be multiplied by −1. Note that if no prefix operator is present, it is treated as if the prefix operator were plus. Some examples of prefix operators with constants and variables are:

```
10  −6  +3
+A  −W
```

The *infix* arithmetic operators are used for calculations and include: addition (+), subtraction (−), multiplication (*), division (/), and exponentiation (∧). An infix operator must appear between each numeric constant and/or variable in a numeric expression. Note that multiplication cannot be implied by simply placing variables side by side or by using parentheses. You must use the multiplication operator (*).

Infix and prefix operators may be entered side by side within a numeric expression. The operators are evaluated in the normal way.

Examples:

```
>NEW

>100 A=6
>110 B=4
>120 C=20
>130 D=2
>140 PRINT A*B/2
>150 PRINT C-D*3+6
>160 END
>RUN
  12
  20

 ** DONE **
```

```
>PRINT 3+-1
 2

>PRINT 2*-3
 -6

>PRINT 6/-3
 -2
```

Numeric Expressions

In evaluating numeric expressions, TI BASIC uses the standard rules for mathematical hierarchy. These rules are outlined here.

1. All expressions within parentheses are evaluated first according to the hierarchical rules.
2. Exponentiation is performed next in order from left to right.
3. Prefix plus and minus are performed.
4. Multiplications and divisions are then completed.
5. Additions and subtractions are then completed.

Examples:

```
>NEW

>100 A=2
>110 B=3
>120 C=4
>130 PRINT A*(B+2)
>140 PRINT B^A-4
>150 PRINT -C^A;(-C)^A
>160 PRINT 10-B*C/6
>170 END
>RUN
  10
  5.
 -16  16
  8

 ** DONE **
```

Note that 0 ^ 0 is defined to be 1 as in ordinary mathematical usage.

```
>PRINT 0^0
  1
```

In the evaluation of a numeric expression if an *underflow* (see page 37) occurs, the value is simply replaced by zero and the program continues running. If an *overflow* (see page 37) occurs in the evaluation of a numeric expression, the value is replaced by the computer's limit, a warning condition is indicated by the message "WARNING: NUMBER TOO BIG," and the program continues running.

```
>NEW

>100 PRINT 1E-200
>110 PRINT 24+1E-139
>120 PRINT 1E171
>130 PRINT (1E60*1E76)/1E50
>140 END
>RUN
  0
  24

 * WARNING:
   NUMBER TOO BIG IN 120
  9.99999E+**

 * WARNING:
   NUMBER TOO BIG IN 130
  1.E+78

 ** DONE **
```

When evaluation of a numeric expression results in division by zero, the value is replaced by the computer's limit with the same sign as the numerator, the message "WARNING: NUMBER TOO BIG" is displayed, and the program continues running. If the evaluation of the operation of exponentiation results in zero being raised to a negative power, the value is replaced by the positive value of the computer's limit, the message "WARNING: NUMBER TOO BIG" is displayed, and the program continues running. If the evaluation of the operation of exponentiation results in a negative number being raised to a non-integral power, the message "BAD VALUE" is displayed, and the program stops running.

```
>NEW

>100 PRINT -22/0
>110 PRINT 0^-2
>120 PRINT (-3)^1.2
>130 END
>RUN

 * WARNING:
   NUMBER TOO BIG IN 100
  -9.99999E+**

 * WARNING:
   NUMBER TOO BIG IN 110
  9.99999E+**

 * BAD VALUE IN 120
```

Relational Expressions

Relational expressions are normally used in the IF-THEN-ELSE statement (see page 79) but may be used anywhere numeric expressions are allowed. When you use relational expressions within a numeric expression, a numeric value of −1 is given if the relation is true and a numeric value of 0 is given if the relation is false.

Relational operations are performed from left to right *before* string concatenation and *after* all arithmetic operations within the expression are completed. To perform string concatenation before relational operations and/or to perform relational operations before arithmetic operations, you must use parentheses. Valid relational operators are:

■ Equal to (=)	■ Not equal to (<>)
■ Less than (<)	■ Less than or equal to (< =)
■ Greater than (>)	■ Greater than or equal to (> =)

An explanation of how string comparisons are performed to give you a true or false result is given on page 79. Remember that the result you obtain from the evaluation of a relational operator is always a number. If you try to use the result as a string, you will get an error.

Examples:

```
>NEW

>100 A=2<5
>110 B=3<=2
>120 PRINT A;B
>130 END
>RUN
 -1  0

 ** DONE **
```

```
>NEW

>100 A$="HI"
>110 B$=" THERE!"
>120 PRINT (A$&B$)="HI!"
>130 END
>RUN
  0

 ** DONE **

>120 PRINT (A$&B$)>"HI"
>RUN
 -1

 ** DONE **

>120 PRINT (A$>B$)*4
>RUN
 -4

 ** DONE **
```

```
>NEW

>100 A=2<4*3
>110 B=A=0
>120 PRINT A;B
>130 END
>RUN
 -1  0

 ** DONE **
```

String Expressions

String expressions are constructed from string variables, string constants, and function references using the operation for concatenation (&). The operation of concatenation allows you to combine strings together. All functions referenced in a string expression must be either functions supplied in TI BASIC (see Built-In String Functions, page 125) or defined by a DEF statement (see page 131) and must have a string value. If evaluation of a string expression results in a value which exceeds the maximum string length of 255 characters, the string is truncated on the right, and the program continues running. No warning is given.

Examples:

```
>NEW

>100 A$="HI"
>110 B$="HELLO THERE!"
>120 C$="HOW ARE YOU?"
>130 MSG$=A$&SEG$(B$,6,7)
>140 PRINT MSG$&" "&C$
>150 END
>RUN
 HI THERE! HOW ARE YOU?

 ** DONE **
```

Reserved Words

Reserved words are words that may not be used as variable names in TI BASIC. Note that only the exact word shown is reserved. You may use reserved words as part of a variable name (for example, ALEN and LENGTH are allowed). The following is a complete list of all reserved words in TI BASIC:

ABS
APPEND
ASC
ATN
BASE
BREAK
BYE
CALL
CHR$
CLOSE
CON
CONTINUE
COS
DATA
DEF
DELETE
DIM
DISPLAY
EDIT
ELSE
END
EOF
EXP
FIXED
FOR
GO
GOSUB
GOTO
IF
INPUT
INT
INTERNAL
LEN
LET
LIST
LOG
NEW
NEXT
NUM
NUMBER
OLD
ON
OPEN
OPTION
OUTPUT
PERMANENT
POS
PRINT
RANDOMIZE
READ
REC
RELATIVE
REM
RES
RESEQUENCE
RESTORE
RETURN
RND
RUN
SAVE
SEG$
SEQUENTIAL
SGN
SIN
SQR
STEP
STOP
STR$
SUB
TAB
TAN
THEN
TO
TRACE
UNBREAK
UNTRACE
UPDATE
VAL
VARIABLE

Statements Used as Commands

Many statements in TI BASIC can be entered as commands with no line number. When a statement is entered as a command, it is executed immediately in the normal way (unless there is an error). The following statements may be entered as commands. The page number on which that statement is discussed in this manual is given in parentheses.

CALL	(100-116)
CLOSE	(149)
DIMension	(136)
DISPLAY	(98)
END	(75)
LET (assignment)	(73)
OPEN	(145)
PRINT	(93, 157)
RANDOMIZE	(121)
REMark	(74)
READ	(89)
RESTORE	(92, 162)
STOP	(76)

Commands Used as Statements

Some commands in TI BASIC may be entered as part of a program. Generally, the commands work the same way when they are used as a statement. The following commands may be used in a program. The page number on which that command is discussed in this manual is given in parentheses.

BREAK (58)
UNBREAK (61)
TRACE (64)
UNTRACE (65)
DELETE (71)

Commands

Introduction

Whenever the prompt and flashing cursor (>□) appear at the bottom of your screen, your computer is in Command (Immediate) Mode. When your computer is in Command Mode, you may enter any of the commands discussed in this section. Commands may be typed in and entered without being preceded by a line number. When a command is entered, your computer performs the required task immediately. Many statements may also be entered as commands. For a complete list of these statements, see page 45. Some of the commands discussed here may be entered as statements. If the command may be entered as a statement, it will be noted in the discussion.

NEW

NEW

The NEW command erases the program that is currently stored in memory. Entering the NEW command cancels the effect of the BREAK command (see page 58) and the TRACE command (see page 64). The NEW command also closes any open files (see OPEN statement, page 145) and releases all space that had been allocated for special characters. In addition, the NEW command erases all variable values and the table in which variable names are stored. After the NEW command is performed, the screen is cleared and the message "TI BASIC READY" is displayed on the screen. The prompt and flashing cursor (>□) indicate that you may enter another command or enter a program line.

Examples:

```
 TI BASIC READY

>□
```

LIST

LIST { [*line-list*]
"*device-name*"[:*line-list*] }

Examples:

When the LIST command is entered, the program lines specified by the *line-list* are displayed. If a *device-name* is entered, then the specified program lines are printed on the specified device. *Device-names* for possible future accessory devices will be given in their respective manuals. If no *device-name* is entered, the specified lines are displayed on the screen.

If the LIST command is entered with no *line-list*, then the entire program is displayed. The program lines are always listed in ascending order. Note that all unnecessary blank spaces that were present when you entered the program line were deleted when the computer accepted the line. Notice that when you list the lines, unnecessary blank spaces have been deleted.

```
>NEW

>100 A=279.3
>120 PRINT A;B
>110 B=-456.8
>130 END
>LIST
 100 A=279.3
 110 B=-456.8
 120 PRINT A;B
 130 END
```

If the *line-list* is entered, it may consist of a single number, a single number preceded by a hyphen (for example: -10), a single number followed by a hyphen (for example: 10-), or a hyphenated range of line numbers. If the *line-list* is:

- A single number — only the program line for the line number specified is displayed on the screen.

```
>LIST 110
 110 B=-456.8
```

- A single number preceded by a hyphen — all program lines with line numbers less than or equal to the line number specified are displayed.

```
>LIST -110
 100 A=279.3
 110 B=-456.8
```

- A single number followed by a hyphen — all program lines with line numbers greater than or equal to the line number specified are displayed.

```
>LIST 120-
 120 PRINT A;B
 130 END
```

- A hyphenated range of line numbers — all program lines with line numbers not less than the first line number in the range and not greater than the second line number are displayed.

```
>LIST 90-120
 100 A=279.3
 110 B=-456.8
 120 PRINT A;B
```

LIST

If there is a program in memory but there are no program lines within the range specified by the *line-list*, then a program line is displayed according to the following rules. If the *line-list* specifies

- Line numbers greater than any in the program – the highest numbered program line is displayed.
- Line numbers less than any in the program – the lowest numbered program line is displayed.
- Line numbers between lines in the program – the next higher numbered line is displayed.

If you enter a LIST command and specify a line number which is equal to zero or greater than 32767, the message "BAD LINE NUMBER" is displayed.

If you specify a line number which is not an integer, the message "INCORRECT STATEMENT" is displayed.

If no program is in memory when you enter a LIST command, the message "CAN'T DO THAT" is displayed.

When program lines are being displayed after the LIST command has been entered, you can stop the listing by pressing the Break key (**SHIFT C**).

Here is a quick summary of the lines listed when specified in the *line-list*.

Command	*Lines Displayed*
LIST	All program lines
LIST x	Program line number x
LIST x-y	Program lines between x and y, inclusive
LIST x-	Program lines greater than or equal to x
LIST -y	Program lines less than or equal to y

Examples:

```
>LIST 150-
 130 END

>LIST -90
 100 A=279.3

>LIST 105
 110 B=-456.8

>LIST 0

 * BAD LINE NUMBER

>LIST 33961

 * BAD LINE NUMBER

>LIST 32.7

 * INCORRECT STATEMENT

>NEW

>LIST

 * CAN'T DO THAT
```

RUN

RUN [line-number]

Entering the RUN command causes the program stored in memory to begin running. Before the program starts running, the values of all numeric variables are set to zero, the values of all string variables are set to null (a string of zero characters), and any space previously allocated for special graphics characters is released.

If no *line-number* is specified when the RUN command is entered, then the program starts running at the lowest numbered line in the program.

If a *line-number* is specified when the RUN command is entered, then the program starts running at the specified program line. Note in this example that since the program begins running at line 110, the value of A remains zero.

If you specify a *line-number* which is not in the program, the message "BAD LINE NUMBER" is displayed.

If you enter a RUN command when there is no program in memory, the message "CAN'T DO THAT" is displayed.

Examples:

```
>NEW

>100 A=-16
>110 B=25
>120 PRINT A;B
>130 END
>RUN
 -16  25

 ** DONE **
```

```
>RUN 110
  0  25

 ** DONE **
```

```
>RUN 115

 * BAD LINE NUMBER
```

```
>NEW

>RUN

 * CAN'T DO THAT
```

BYE

BYE

When you are finished working and are ready to leave BASIC, simply enter the BYE command. We recommend that you always use the BYE command (instead of **SHIFT Q**) when you wish to leave BASIC. When the BYE command is entered, the first job your computer performs is closing all open files (see OPEN statement, page 145). Then, the program in memory and all variable values are erased. Finally, the computer is reset so that it is ready to go again when you want to return to BASIC. After the BYE command is performed, the master computer title reappears.

Examples:

```
>NEW

>100 LET X$="HELLO, GENIUS!"
>110 PRINT X$
>120 END
>RUN
 HELLO, GENIUS!

 ** DONE **

>BYE

--master computer title
  screen appears
```

NUMBER

{ **NUMBER** }
{ **NUM** } [*initial-line*][,*increment*]

Examples:

When the NUMBER command is entered, your computer automatically generates line numbers for you. Your computer is in Number Mode when it is generating line numbers. In Number Mode each line entered in response to a generated line number is added to the program.

The first line number displayed after entering the NUMBER command is the specified *initial-line*. Succeeding line numbers are generated using the specified *increment*. To terminate the automatic generation of line numbers and leave Number Mode, press **ENTER** immediately after the generated line number is displayed. The empty line is not added to the program.

```
>NEW

>NUMBER 10,5

>10 C=38.2
>15 D=16.7
>20 PRINT C;D
>25 END
>30 ENTER
>LIST
 10 C=38.2
 15 D=16.7
 20 PRINT C;B
 25 END
```

If no *initial-line* and no *increment* are specified, then 100 is used as the *initial-line* and 10 is used as the *increment*.

```
>NEW

>NUM
>100 B$="HELLO!"
>110 PRINT B$
>120 END
>130 ENTER
```

If you specify only an *initial-line*, then 10 is used as the *increment*.

```
>NEW

>NUMBER 50
>50 C$="HI!"
>60 PRINT C$
>70 END
>80 ENTER
```

If you specify just an *increment*, then 100 is used as the *initial-line*. Note the comma before the five in the example. Remember, if you wish to specify only an *increment*, the comma must be typed before the *increment*.

```
>NEW

>NUM ,5
>100 Z=99.7
>105 PRINT Z
>110 END
>115 ENTER
```

NUMBER

When you are in Number Mode, if a line number generated is already a line in the program, then the existing program line is displayed with the line number. Note that when an existing program line is displayed in Number Mode, the prompt character (>) is not shown to the left of the line number. This indicates the line is an existing program line and you may choose to edit the line. For information on editing, see the section below. If you do not want to change the existing line, simply press **ENTER** when the line is displayed and it will not be changed. After you press **ENTER**, the next line number is generated.

Examples:

```
>NEW

>100 A=37.1
>110 B=49.6
>NUMBER 110
 110 B=49.6
>120 PRINT A;B
>130 END
>140 ENTER
>LIST
 100 A=37.1
 110 B=49.6
 120 PRINT A;B
 130 END
```

In Number Mode, if you enter a program line and an error occurs, the appropriate error message is displayed as usual and then the same line number is displayed again. Retype the line correctly and then enter it again. If a line number would be generated in Number Mode which is greater than 32767, the computer leaves Number Mode.

Editing in Number Mode

Whether you are entering new lines or changing existing program lines while in Number Mode, all of the special editing keys may be used. Since some of the keys work differently in Number Mode than in Command Mode (see page 47), the keys and how they work in Number Mode are discussed here.

ENTER – This key has different functions depending on the situation. The functions and situations are described below.

- If you press **ENTER** immediately after a line number is generated, then the computer leaves Number Mode.
- If you type in a statement after the line number is generated and then press **ENTER**, the new line is added to the program. Then the next line number is generated.
- If an existing program line is displayed and you press **ENTER** immediately after it is displayed, the line remains the same in the program. Then the next line number is generated.
- If an existing program line is displayed and you erase the entire text of the line (leaving only the line number on the screen) and then press **ENTER** , the computer leaves Number Mode. The program line is not removed from the program.
- If you edit a line after it is displayed as an existing program line and text still remains after the line number and then press **ENTER**, the existing program line is replaced by the edited line. Then the next line number is generated.

NUMBER

SHIFT [↑] **(UP) – The Up-Arrow key works exactly the same as the ENTER key in Number Mode.**

SHIFT [↓] (DOWN) – The Down-Arrow key works exactly the same as the **ENTER** key in Number Mode.

SHIFT [←] (LEFT) – The Left-Arrow key moves the cursor one position to the left. When the cursor moves over a character it does not delete or change it in any way.

SHIFT [→] (RIGHT) – The Right-Arrow key moves the cursor one position to the right. Using this key allows you to move the cursor over a character without deleting or changing it in any way.

SHIFT G (INS) The Insert key works in Number Mode just as it does in Command Mode. See Special Keys, page 33 for information.

SHIFT F (DEL) – The Delete key works in Number Mode just as it does in Command Mode. See Special Keys, page 34 for information.

SHIFT C (CLEAR) – If you press the Clear key at any time while in Number Mode, the current line scrolls up on the screen and the computer leaves Number Mode. Any changes which had been made on the line before you pressed the Clear key are ignored. Thus, if you were editing an existing program line, the program line does not change. If you were typing in a line, the line is not added to the program.

SHIFT T (ERASE) – The Erase key erases the entire text of the program line being displayed. The line number is still displayed.

RESEQUENCE

{RESEQUENCE} [*initial-line*][,*increment*]
{RES}

When the RESEQUENCE command is entered, all lines in the program are assigned new line numbers according to the specified *initial-line* and *increment.*

The new line number of the first line in the program is the specified *initial-line.* Succeeding line numbers are assigned using the specified *increment.*

If no *initial-line* and no *increment* are specified, then 100 is used as the *initial-line* and 10 is used as the *increment.*

If you specify only an *initial-line* then 10 is used as the *increment.*

If you specify just an *increment,* then 100 is used as the *initial-line.* Note the comma before the five in the example. Remember, if you wish to specify only an *increment,* the comma must be typed in before the *increment.*

All line number references in TI BASIC statements contained in the program are changed to the new line numbers. Line numbers which may be mentioned in the REM statement (see page 74) are not changed since they are not essential to the running of the program.

Examples:

```
>NEW

>100 A=27.9
>110 B=34.1
>120 PRINT A;B
>130 END

>RESEQUENCE 20,5
>LIST
 20 A=27.9
 25 B=34.1
 30 PRINT A;B
 35 END

>RES
>LIST
 100 A=27.9
 110 B=34.1
 120 PRINT A;B
 130 END

>RES 50
>LIST
 50 A=27.9
 60 B=34.1
 70 PRINT A;B
 80 END

>RES ,5
>LIST
 100 A=27.9
 105 B=34.1
 110 PRINT A;B
 115 END


>NEW

>100 REM THE VALUE OF "A" WIL
 L BE PRINTED IN LINE 120
>110 A=A+1
>120 PRINT A
>130 GO TO 110
>RESEQUENCE 10,5
>LIST
 10 REM THE VALUE OF "A" WIL
 L BE PRINTED IN LINE 120
 15 A=A+1
 20 PRINT A
 25 GO TO 15
```

RESEQUENCE

If a line number is used in a program line which is not a currently used line number, then the line number reference is changed to 32767. No error or warning is given.

If you enter a value for the *initial-line* and *increment* which would give values greater than 32767 for some new line numbers, the message "BAD LINE NUMBER" is displayed. If this error occurs, no line numbers in the program are changed.

If you enter a RESEQUENCE command while no program is in memory, the message "CAN'T DO THAT" is displayed.

Examples:

```
>NEW

>100 Z=Z+2
>110 PRINT Z
>120 IF Z=50 THEN 150
>130 GO TO 100
>140 END
>RES 10,5
>LIST
 10 Z=Z+2
 15 PRINT Z
 20 IF Z=50 THEN 32767
 25 GO TO 10
 30 END
```

```
>RESEQUENCE 32600,100
 * BAD LINE NUMBER

>LIST
 10 Z=Z+2
 15 PRINT Z
 20 IF Z=50 THEN 32767
 25 GO TO 10
 30 END
```

```
>NEW

>RESEQUENCE

 * CAN'T DO THAT
```

BREAK

BREAK *line-list*

When the BREAK command is entered, breakpoints are set at the program lines listed in the *line-list*. Breakpoints are usually set to help you find errors in your program. When you set a breakpoint at a specific line using the BREAK command, you tell the computer to stop running the program before performing the statement on that line.

The *line-list* is a list of line numbers where you wish to set breakpoints. The line numbers are separated by commas (for example: BREAK 10,23,35). Of course, you may choose to have only one line number in the list.

Each time a line where a breakpoint is set is reached while the program is running, the program stops running before the statement on that line is performed. When the program stops running because of a breakpoint, the message "BREAKPOINT AT line-number" is displayed, and you are prompted with the flashing cursor to enter a command.

When the program stops running because of a breakpoint, you may enter any command or any statement that can be used as a command (see page 45). There is no change in the value of the variables unless you enter a statement that will assign a new value. Note that in this example C still equals zero since the assignment in statement 110 has not been performed.

You can start running the program again (beginning with the line where the breakpoint was set) by entering the CONTINUE command (see page 63). Note the value of A was changed earlier in the example. You cannot enter the CONTINUE command after you have edited the program (added, deleted, or changed program lines). This prevents errors that could result from starting a revised program in the middle. If you enter a CONTINUE command after you have edited the program, the message "CAN'T CONTINUE" is displayed on the screen.

Examples:

```
>NEW

>100 A=26.7
>110 C=19.3
>120 PRINT A
>130 PRINT C
>140 END

>BREAK 110

>RUN

 * BREAKPOINT AT 110

>□

>LIST 110
 110 C=19.3
>PRINT A;C
  26.7  0

>A=5.8

>PRINT A
  5.8

>CONTINUE
  5.8
  19.3

 ** DONE **

>BREAK 120

>RUN

 * BREAKPOINT AT 120

>110 ENTER
>CONTINUE
 * CAN'T CONTINUE
```

BREAK

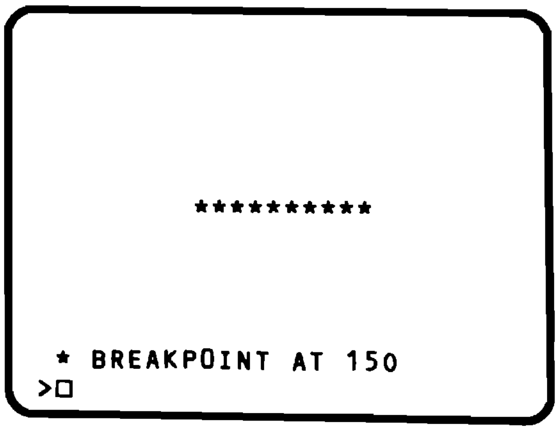
When a breakpoint is taken (program stops running because of a breakpoint), the breakpoint at that line is removed. Another way to remove breakpoints is to use the UNBREAK command (see page 61). If a breakpoint is set at a program line and that line is deleted, the breakpoint is also removed. Breakpoints are removed from all program lines when a SAVE command (see page 68) or a NEW command (see page 48) is entered. Note that in the example, the breakpoint at 110 was removed when the breakpoint was taken, while the breakpoint at 130 was removed by the UNBREAK command.

Examples:

```
>110 C=19.3
>RUN
  26.7
  19.3

 ** DONE **

>BREAK 110,130

>RUN

 * BREAKPOINT AT 110

>UNBREAK

>CONTINUE
  26.7
  19.3

 ** DONE **

>RUN
  26.7
  19.3

 ** DONE **
```

Whenever a breakpoint is taken, the standard character set (see page 163) is stored. Thus, any standard characters that had been redefined by CALL CHAR (see page 104) will be converted back to the standard characters. Characters defined in the range 96-159 are unaffected. Note that when this example program is run, a solid bar appears on the screen until the breakpoint is taken. When the breakpoint is taken, the bar becomes a row of asterisks (*) since character 42 is a standard character.

```
>NEW

>100 CALL CLEAR
>110 CALL CHAR(42,"FFFFFFFFFF
 FFFFFF")
>CALL HCHAR(12,12,42,10)
>130 FOR I=1 TO 500
>140 NEXT I
>150 END
>BREAK 150

>RUN

 --screen clears

 --solid black line appears
   on screen
```



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



 * BREAKPOINT AT 150
>□
```

```
>CONTINUE

 ** DONE **
```

BREAK

The BREAK command may also be used as a statement in programs. If the BREAK command is entered as a statement with a *line-list*, then breakpoints are set at the line numbers specified. Breakpoints set in this manner may be removed as discussed earlier. Remember, though, when the BREAK command is entered as a statement with a *line-list*, the breakpoints are set again each time the statement is performed.

Examples:

```
>NEW

>100 B=29.7
>110 BREAK 120,140
>120 H=15.8
>130 PRINT B
>140 PRINT H
>150 END
>RUN

 * BREAKPOINT AT 120

>UNBREAK

>CONTINUE
  29.7
  15.8

 ** DONE **
```

If the BREAK command is entered as a statement and no *line-list* is specified, then the statement itself acts like a breakpoint. Each time the statement is performed, the program stops running. The only way to keep the program from stopping at a BREAK statement is to delete the line from the program. Note that a BREAK command without a *line-list* may only be entered as a program line.

```
>110 BREAK
>RUN

 * BREAKPOINT AT 110

>CONTINUE
  29.7
  15.8

 ** DONE **

>110 ENTER
```

If you specify a line number in the *line-list* which is equal to zero or greater than 32767, the message "BAD LINE NUMBER" is displayed and the command is ignored (no breakpoints are set at any line specified).

```
>BREAK 120,130140

 * BAD LINE NUMBER

>RUN
  29.7
  15.8

 ** DONE **
```

If you specify a line number in the *line-list* which is a valid line number but is not a line in the program, the warning "BAD LINE NUMBER" is displayed. Breakpoints will be set at the lines specified which are program lines.

```
>110 BREAK 125,140
>RUN

 * WARNING:
   BAD LINE NUMBER IN 110
  29.7

 * BREAKPOINT AT 140

>CONTINUE
  15.8

 ** DONE **
```

UNBREAK

UNBREAK [*line-list*]

The UNBREAK command is used to remove breakpoints from the program lines listed in the *line-list*. For an explanation of breakpoints and how they are set, see the BREAK command, page 58.

The *line-list* is a list of line numbers where you want to remove breakpoints. The line numbers are separated by commas. (For example: UNBREAK 10,23.) If you specify only one line number in the *line-list*, no commas are needed.

If you enter an UNBREAK command with no *line-list*, then all breakpoints which have been set by a BREAK command or statement are removed. Note that the UNBREAK command has no effect on a BREAK statement with no *line-list*. The only way to keep the program from stopping at a BREAK statement with no *line-list* is to delete the line.

The UNBREAK command may also be used as a statement in a program. The UNBREAK statement is performed just like the UNBREAK command. Note in the example, the UNBREAK statement removed the breakpoint that was set at 130.

Examples:

```
>NEW

>100 A=26.7
>110 C=19.3
>120 PRINT A
>130 PRINT C
>140 END
>BREAK 110,130

>RUN

 * BREAKPOINT AT 110

>UNBREAK 130

>CONTINUE
  26.7
  19.3

 ** DONE **
```

```
>125 BREAK
>BREAK 100,120,130

>RUN

 * BREAKPOINT AT 100
>UNBREAK

>CONTINUE
  26.7

 * BREAKPOINT AT 125

>CONTINUE
  19.3

 ** DONE **
```

```
>BREAK 130

>125 UNBREAK 130
>RUN
  26.7
  19.3

 ** DONE **

>125 ENTER
```

UNBREAK

If you specify a line number in the *line-list* which is equal to zero or greater than 32767, the message "BAD LINE NUMBER" is displayed and the command is ignored (no breakpoints are removed at any line specified).

If you specify a line number in the *line-list* which is a valid line number but is not a line in the program, the warning "BAD LINE NUMBER" is displayed. Breakpoints are removed at the lines specified which are program lines.

Examples:

```
>BREAK 130

>UNBREAK 130,110150

 * BAD LINE NUMBER

>RUN
  26.7

 * BREAKPOINT AT 130

>CONTINUE
  19.3

 ** DONE **
```

```
>BREAK 130

>UNBREAK 130,105

 * WARNING:
   BAD LINE NUMBER

>RUN
  26.7
  19.3

 ** DONE **
```

CONTINUE

{CONTINUE}
{CON}

The CONTINUE command may be entered whenever the program stops running because of a breakpoint. For an explanation of breakpoints and how they are set, see the BREAK command, page 58. Remember that a breakpoint is also taken when the Break key (SHIFT C) is pressed while the program is running.

You cannot enter the CONTINUE command when the program has stopped running for a breakpoint if you have edited the program (added, deleted, or changed program lines). This prevents errors that could result from starting a revised program in the middle. If you enter a CONTINUE command after you have edited the program, the message "CAN'T CONTINUE" is displayed on the screen.

Whenever a breakpoint is taken, the standard character set (see page 163) is stored. Thus, any standard characters that had been redefined by CALL CHAR (see page 104) will be converted back to the standard characters. Characters defined in the range 96-159 are unaffected. If you continue execution after a breakpoint, the standard character set is used. Note in the example that character 42 was defined in statement 110 to be a solid block; however, when the breakpoint was taken, it was changed back to its standard character, an asterisk (*). The triangle defined for character code 96 is unaffected by the breakpoint.

Examples:

```
>NEW

>100 A=9.6
>110 PRINT A
>120 END
>BREAK 110

>RUN

 * BREAKPOINT AT 110

>CONTINUE
  9.6

 ** DONE **

>BREAK 110

>RUN

 * BREAKPOINT AT 110

>100 A=10.1
>CONTINUE
 * CAN'T CONTINUE
```

```
>NEW

>100 CALL CLEAR
>110 CALL CHAR(42,"FFFFFFFFFF
 FFFFFF")
>120 CALL CHAR(96,"0103070F1F
 3F7FFF")
>130 CALL HCHAR(10,10,42,5)
>140 CALL HCHAR(11,10,96,5)
>150 FOR I=1 TO 500
>160 NEXT I
>170 END
>BREAK 130

>RUN

 * BREAKPOINT AT 130

>CONTINUE

       *****
       ◢◢◢◢◢

 ** DONE **
```

TRACE

TRACE

The TRACE command allows you to see the order in which the computer performs statements as it runs a program. After the TRACE command is entered, the line number of each program line is displayed before the statement is performed. The TRACE command is most often used to help find errors, such as unwanted infinite loops, in a program.

The TRACE command may be placed as a statement in a program. The effect of the TRACE command or statement is cancelled when the NEW command or UNTRACE command or statement is performed.

Examples:

```
>NEW

>100 PRINT "HI"
>110 B=27.9
>120 PRINT :B
>130 END
>TRACE

>RUN
 <100>HI
 <110><120>
  27.9
 <130>
 ** DONE **
```

```
>UNTRACE

>105 TRACE
>RUN
 HI
 <110><120>
  27.9
 <130>
 ** DONE **
```

UNTRACE

UNTRACE

The UNTRACE command cancels the effect of the TRACE command. The UNTRACE command may be used as a statement in a program.

Examples:

```
>NEW

>100 FOR I=1 TO 2
>110 PRINT I
>120 NEXT I
>130 END
>TRACE

>RUN
 <100><110> 1
 <120><110> 2
 <120><130>
 ** DONE **

>UNTRACE

>RUN
  1
  2

 ** DONE **
```

EDIT

$$\left\{\begin{array}{l}\text{EDIT } \textit{line-number}\\ \textit{line-number} \left\{\begin{array}{l}\textbf{SHIFT } \boxed{\uparrow}\\ \textbf{SHIFT } \boxed{\downarrow}\end{array}\right\}\end{array}\right\}$$

Existing program lines may be changed by entering Edit Mode. You can enter Edit Mode by entering the EDIT command followed by a *line-number* or by typing in a *line-number* followed by **SHIFT** ↑ (Up-Arrow) or **SHIFT** ↓ (Down-Arrow). Either way you choose to enter Edit Mode will bring the line specified by the *line-number* onto the screen. If you specify a *line-number* which is not in the program, the message "BAD LINE NUMBER" is displayed.

When you enter Edit Mode, the program line you requested is displayed on the screen. The prompt character (>) is not displayed to the left of the line when you are in Edit Mode. When the requested line is displayed, the flashing cursor is positioned in the second character position to the right of the line number. Changes may be made to any character on the line except the line number using the special keys described below and typing over the characters you wish to change. You cannot move the cursor back over the line number. Thus, you cannot change the line number in Edit Mode. The special editing keys and their functions in Edit Mode are discussed here.

ENTER – When you press the **ENTER** key, all changes you have made to the program line become permanent and the computer leaves Edit Mode. If you have erased the entire text of the program line and then press **ENTER**, the program line is deleted. Note that the cursor does not have to be at the end of the line for the entire line to be entered.

SHIFT ↑ (UP) – When you press the Up-Arrow key, all changes you have made to the program line are entered and become permanent. The next lower numbered line in the program is then displayed for editing. If no lower numbered program line exists, then the computer leaves Edit Mode. Note that the cursor does not have to be at the end of the line for the entire line to be entered by the Up-Arrow key.

SHIFT ↓ (DOWN) – When you press the Down-Arrow key, all changes you have made to the program line are entered and become permanent. The next higher numbered program line is then displayed for editing. If no higher numbered program line exists, then the computer leaves Edit Mode. Note that the cursor does not have to be at the end of the line for the entire line to be entered by the Down-Arrow key.

EDIT

SHIFT ← **(LEFT) – The Left-Arrow (backspace) key moves the cursor one position to the left. When the cursor moves over a character it does** not delete or change it in any way.

SHIFT → (RIGHT) – The Right-Arrow (forwardspace) key moves the cursor one position to the right. Using this key allows you to move the cursor over a character without deleting or changing it in any way.

SHIFT G (INS) – The Insert key works in Edit Mode just as it does in Command Mode. See Special Keys, page 33 for information.

SHIFT F (DEL) – The Delete key works in Edit Mode just as it does in Command Mode. See Special Keys, page 34 for information.

SHIFT C (CLEAR) – If you press the Clear key at any time while in Edit Mode, the current line scrolls up on the screen and the computer leaves Edit Mode. Any changes which had been made on the line before you pressed the Clear key are ignored. Thus, the existing program line does not change.

SHIFT T (ERASE) – The Erase key erases the entire text of the program line currently displayed for editing. The line number is not erased.

SAVE

SAVE *file-name*

The SAVE command allows you to copy the current program in the computer's memory onto an accessory device. By using the OLD command (see page 70), you can later put the program into memory for running or editing.

An explanation of how to use the audio cassette recorders to SAVE a program is given here. As additional accessory devices become available, their accompanying manuals will describe how to use the SAVE command.

You select which cassette recorder the computer will use by entering the *file-name* CS1 or CS2 following the keyword SAVE. After you have connected your recorder to the computer (see pages 15-16 of this book for detailed instructions on attaching cassette recorders to your computer), type the SAVE command, press **ENTER**, and the computer will begin printing instructions on the screen to help you understand the SAVE procedures. Follow the directions as they appear on the screen.

On the right are the computer-generated SAVE instructions. CS1 is used in the example, but the same procedures apply for CS2 also. You can find a more detailed description of these instructions on pages 16-18 of this book.

When you enter the SAVE command, the computer tells you how to use the recorder, as shown on the right. After the program has been copied, the computer asks if you want to check the tape to be sure your program was recorded correctly. If you press **N**, the flashing cursor will appear at the left of the screen. You may then type any BASIC command you wish. If you press **Y**, directions for activating the recorder will appear.

Examples:

```
>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)? Y

 * REWIND CASSETTE TAPE    CS1
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY    CS1
   THEN PRESS ENTER

 * CHECKING

 * DATA OK

 * PRESS CASSETTE STOP    CS1
   THEN PRESS ENTER
```

SAVE

If an error occurred, you may choose one of these three options:

- **Press R** to record your program again. The same instructions listed previously will guide you.
- **Press C** to repeat the checking procedures. At this point you may wish to adjust the recorder volume and/or tone controls.
- **Press E** to "exit" from the recording procedure. The computer will tell you to stop the cassette and press **ENTER**. You will see an error message on the screen. This means that the SAVE routine did not properly record your program. After checking your recorder, you can try to record the program again. When the flashing cursor reappears on the screen, enter any BASIC command you wish.

See page 17 for a list of items to check if an error occurred while saving your program.

When the SAVE command is performed, whether or not an error occurred in recording, the program remains in memory.

Examples:

```
* ERROR - NO DATA FOUND
  PRESS R TO RECORD
  PRESS C TO CHECK
  PRESS E TO EXIT

           or

* ERROR IN DATA DETECTED
  PRESS R TO RECORD
  PRESS C TO CHECK
  PRESS E TO EXIT

* I/O ERROR 66
```

OLD

OLD *file-name*

The OLD command copies a previously SAVEd program into the computer's memory. You can then run, list, or change the program. An explanation for using the audio cassette tape recorder (CS1) with the OLD command is given here. As additional accessory devices become available, their accompanying manuals will describe how to use the OLD command.

After you type the OLD command and press **ENTER**, the computer will begin printing instructions on the screen to help you through the procedures. Follow the directions as they appear on the screen. Be sure you have connected the recorder and inserted the proper cassette tape. (See pages 15-16 of this book for detailed instructions on attaching a cassette recorder to your computer.)

On the right are the instructions displayed on the screen when you enter the OLD command. You will find a detailed description of these procedures on pages 16-18.

If the computer did not successfully read your program into memory, an error occurs and you may choose either of these options:

- Press **R** to repeat the reading procedure. Before repeating the procedure, be sure to check the items listed on page 18.
- Press **E** to "exit" from the reading procedure. An error message indicating that the computer did not properly read your program into memory is displayed.

When the flashing cursor reappears on the screen, you may enter any BASIC command you wish.

Examples:

```
>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

            or

 * ERROR - NO DATA FOUND
   PRESS R TO READ
   PRESS E TO EXIT

 * I/O ERROR 56
```

DELETE

DELETE { *file-name* }
       { *program-name* }

The DELETE command allows you to remove a program or a data file from the computer's filing system. The *file-name* and *program-name* are string expressions. If a string constant is used, you must enclose it in quotes.

You may also remove data files from the computer system by using the keyword DELETE in the CLOSE statement (see page 149). The action performed depends upon the device used.

If you use DELETE with cassette tape recorders, no action occurs. The message on the right will appear on the screen.

Examples:

```
>SAVE NAME$

>DELETE NAME$


>500 CLOSE #7:DELETE


>DELETE "CS1"

 * PRESS CASSETTE STOP      CS1
   THEN PRESS ENTER
```

General Program Statements

Introduction

This section describes those general program statements that do not serve an input-output function. They include the LET statement, which allows you to assign values to variables, the STOP, END, and REMark statements, and those statements which control the path the computer takes when it runs your program. These program control statements, including the GOTO, the ON-GOTO, the IF-THEN-ELSE, the FOR-TO-STEP, and the NEXT statements, allow you to easily program loops and conditional and unconditional branches. By using the statements in this section and in the Input-Output section, you can write enjoyable, useful programs.

LET (Assignment Statement)

[LET] *variable* = *expression*

The LET statement allows you to assign values to variables in your program. The computer evaluates the *expression* to the right of the equals sign and puts its value into the *variable* specified to the left of the equals sign.

The *variable* and the *expression* must correspond in type: numeric expressions must be assigned to numeric variables (see page 39); string expressions must be assigned to string variables (see page 39). The rules governing overflow and underflow for the evaluation of a numeric expression are used in the LET statement. See page 37 for a full explanation. When a string expression is evaluated, if the string length exceeds 255 characters, then the string is truncated on the right, and the program continues. No warning is given.

You may use relational operators in numeric and string expressions. The result of a relational operator is -1 if the relationship is true and is 0 if the relationship is false (see page 42 for a complete explanation).

Examples:

```
>NEW

>100 LET M=1000
>110 LET C=186000
>120 E=M*C^2
>130 PRINT E
>140 END
>RUN
  3.4596E+13

 ** DONE **
```

```
>NEW

>100 LET X$="HELLO, "
>110 NAME$="GENIUS!"
>120 PRINT X$;NAME$
>130 END
>RUN
 HELLO, GENIUS!

 ** DONE **
```

```
>NEW

>100 LET A=20
>110 B=10
>120 LET C=A>B
>130 PRINT A;B;C
>140 C=A<B
>150 PRINT A;B;C
>160 END
>RUN
  20   10 -1
  20   10  0

 ** DONE **
```

REMark

REM *remark*

The REMark statement allows you to explain and document your program by inserting comments in the program itself. When the computer encounters a REMark statement while running your program, it takes no action but proceeds to the next statement.

You may use any printable character in a REMark statement. The length of the REMark statement is limited by the length of the input line (112 characters or four lines on the screen). If you do not wish to break a word in the middle, press the space bar repeatedly until the cursor returns to the left side of the screen, and then you may begin typing again.

Examples:

```
>NEW

>100 REM COUNTING FROM 1 TO
 10
>110 FOR X=1 TO 10
>120 PRINT X;
>130 NEXT X
>140 END
>RUN
  1  2  3  4  5  6  7  8  9
  10
 ** DONE **
```

```
>NEW

>100 A=762
>110 B=425
>120 REM NOW PRINT THE SUM OF
 A AND B
>130 PRINT A+B
>140 END
>RUN
  1187

 ** DONE **
```

END

END

The END statement terminates your program when it is being run and may be used interchangeably with the STOP statement in TI BASIC. Although the END statement can appear anywhere in the program, it is normally placed at the last line number in the program and thus ends the program both physically and logically. Although you may place END statements anywhere in your program, the STOP statement is usually used if you want to have other termination points in your program (see page 76). In TI BASIC you are not required to place an END statement in the program.

Examples:

```
>NEW

>100 A=10
>110 B=20
>120 C=A*B
>130 PRINT C
>140 END
>RUN
  200

 ** DONE **
```

STOP

STOP

The STOP statement terminates your program when it is being run and can be used interchangeably with the END statement in TI BASIC. You can place STOP statements anywhere in your program and use several STOP statements in the same program. Many BASIC programmers use the END statement if there is only one ending point in the program (see page 75).

Examples:

```
>NEW

>100 A=5
>110 B$="TEXAS INSTRUMENTS"
>120 PRINT B$;A
>130 STOP
>RUN
 TEXAS INSTRUMENTS 5

 ** DONE **
```

```
>NEW

>100 CALL CLEAR
>110 FOR I=1 TO 15
>120 CALL HCHAR(1,1,42,768)
>130 GOSUB 160
>140 NEXT I
>150 STOP
>160 F=I
>170 B=I+1
>180 CALL COLOR(2,F,B)
>190 RETURN
>200 END
>RUN

  --SCREEN WILL FILL WITH
    ASTERISKS AND CHANGE
    COLORS 15 TIMES

 ** DONE **
```

GOTO

$\left\{\begin{array}{l}\text{GOTO}\\\text{GO TO}\end{array}\right\}$ *line-number*

The GOTO statement allows you to transfer control backward or forward within a program. Whenever the computer reaches a GOTO statement, it will always jump to the statement with the specified *line-number*. This is called an *unconditional* branch.

In the program on the right, line 170 is an *unconditional* branch. The computer will always skip to line 140 at this point. Line 160 is a *conditional* branch (see page 79). The computer will jump to line 180 only if COUNT and DAYS are equal.

If you should tell the computer to skip to a *line-number* that does not exist in your program, the program will stop running and print the message "BAD LINE NUMBER."

Note that the space between the words GO and TO is optional.

Examples:

```
>NEW

>100 REM HOW MANY GIFTS ON
 THE 12 DAYS OF CHRISTMAS?
>110 GIFTS=0
>120 DAYS=1
>130 COUNT=0
>140 COUNT=COUNT+1
>150 GIFTS=GIFTS+1
>160 IF COUNT=DAYS THEN 180
>170 GOTO 140
>180 DAYS=DAYS+1
>190 IF DAYS<=12 THEN 130
>200 PRINT "TOTAL NUMBER OF G
 IFTS IS";GIFTS
>210 END
>RUN
 TOTAL NUMBER OF GIFTS IS 78

 ** DONE **
```

ON-GOTO

ON *numeric-expression* $\left\{\begin{array}{c}\text{GOTO}\\\text{GO TO}\end{array}\right\}$ *line-number*[,*line-number*]. . .

The ON-GOTO statement tells the computer to jump to one of several program lines, depending on the value of the *numeric-expression.*

The computer first evaluates the *numeric-expression* and rounds the result to an integer. This integer then becomes a pointer for the computer, indicating which program line in the ON-GOTO statement to perform next. If the value of the *numeric-expression* is 1, the computer will proceed to the statement with the first *line-number* specified in the ON-GOTO statement. If the value is 2, the computer will branch to the statement with the second *line-number* listed in the ON-GOTO statement, and so on.

If the rounded value of the *numeric-expression* is less than 1 or greater than the number of *line-numbers* listed in the ON-GOTO statement, the program will stop running and print "BAD VALUE IN xx." If the *line-number* you specify is outside the range of line numbers in your program, the message "BAD LINE NUMBER" is displayed and the program stops running.

Examples:

```
>NEW

>100 REM HOW DOES ON-GOTO
 WORK?
>110 INPUT X
>120 ON X GOTO 130,150,170,19
 0,210
>130 PRINT "X=1"
>140 GOTO 110
>150 PRINT "X=2"
>160 GOTO 110
>170 PRINT "X=3"
>180 GOTO 110
>190 PRINT "X=4"
>200 GOTO 110
>210 END
>RUN
 ? 2
 X=2
 ? 1.2
 X=1
 ? 3.7
 X=4
 ? 6

 * BAD VALUE IN 120
```

IF-THEN-ELSE

IF {*relational-expression* / *numeric-expression*} THEN *line-1* [ELSE *line-2*]

The IF-THEN-ELSE statement allows you to change the normal sequence of your program execution by using a *conditional* branch. See page 42 for a full explanation of *relational-expressions*.

The computer evaluates the expression you have included in the statement, such as A>50. If the expression is true, the computer will jump to *line-1*, which follows the word THEN. If the condition is false, the computer will jump to *line-2* following the word ELSE. If ELSE is omitted, the computer continues with the next program line.

The allowable relational operators in TI BASIC are:

- equal to (=)
- less than (<)
- greater than (>)
- not equal to (< >)
- less than or equal to (< =)
- greater than or equal to (> =)

Here are some valid relationship tests:

- A>7
- A$<"YES"
- (A+B)/2< >AVG
- CHR$(L)="A"
- (A$&C$)> =D$

A *numeric-expression* must be compared to another *numeric-expression* and a *string-expression* to another *string-expression*. *Numeric-expressions* are compared algebraically. *String-expressions* are compared left-to-right, character by character, using the ASCII character codes (see Appendix page 163 for ASCII character codes). A character with a lower ASCII code will be considered less than one with a higher ASCII code. Thus, you can sort strings into numeric or alphabetic order. If one string is longer than the other, the comparison is made for each character in the shorter string. If there is no difference, the computer considers the longer string to be greater.

Examples:

```
>NEW

>100 REM FIND THE LARGEST OF
 A SET OF NUMBERS
>110 INPUT "HOW MANY VALUES?"
 :N
>120 INPUT "VALUE?":A
>130 L=A
>140 N=N-1
>150 IF N<=0 THEN 180
>160 INPUT "VALUE?":A
>170 IF L>A THEN 140 ELSE 130

>180 PRINT L;"IS THE LARGEST"

>190 END
>RUN
 HOW MANY VALUES?3
 VALUE?456
 VALUE?321
 VALUE?292
  456 IS THE LARGEST

 ** DONE **
```

```
>NEW

>100 INPUT "A$ IS ":A$
>110 INPUT "B$ IS ":B$
>120 IF A$=B$ THEN 160
>130 IF A$<B$ THEN 180
>140 PRINT "B$ IS LESS"
>150 GOTO 190
>160 PRINT "A$=B$"
>170 GOTO 190
>180 PRINT "B$ IS GREATER"
>190 END
>RUN
 A$ IS TEXAS
 B$ IS TEX
 B$ IS LESS

 ** DONE **

>RUN
 A$ IS TAXES
 B$ IS TEX
 B$ IS GREATER

 ** DONE **
```

IF-THEN-ELSE

An alternative format of the IF-THEN-ELSE statement is to use a *numeric-expression* with no relationship expressed. In the example on the right, the computer will evaluate the expression A + B. If the result is zero, the expression is treated as false. A non-zero result is treated as true. This is the same as:

IF *expression* <> 0 THEN *line-1*.

Examples:

```
>NEW

>100 INPUT "A IS ":A
>110 INPUT "B IS ":B
>120 IF A+B THEN 150
>130 PRINT "RESULT IS ZERO,EX
 PRESSION FALSE"
>140 GOTO 100
>150 PRINT "RESULT IS NON-ZER
 O,EXPRESSION TRUE"
>160 GO TO 100
>RUN
 A IS 2
 B IS 3
 RESULT IS NON-ZERO,EXPRESSIO
 N TRUE
 A IS 2
 B IS -2
 RESULT IS ZERO,EXPRESSION FA
 LSE

 (PRESS SHIFT C TO END LOOP)
```

FOR-TO-STEP

FOR *control-variable*=*initial-value* TO *limit*[STEP *increment*]

The FOR-TO-STEP statement is used for easy programming of repetitive (iterative) processes. Together with the NEXT statement (see page 84) the FOR-TO-STEP statement is used to construct a FOR-NEXT loop. If the STEP clause is omitted, the computer will use an *increment* of +1.

The *control-variable* is a numeric variable which acts as a counter for the loop. When the FOR-TO-STEP statement is performed, the *control-variable* is set to the *initial-value.* The computer then performs program statements until it encounters a NEXT statement.

When the NEXT statement is performed, the computer increments the *control-variable* by the amount specified in the STEP clause. (When the *increment* is a negative value, the *control-variable* is actually reduced by the STEP amount.) The computer then compares the *control-variable* to the value of the *limit.* If the *control-variable* does not yet exceed the *limit,* the computer repeats the statements following the FOR-TO-STEP statement until the NEXT statement is again encountered and performed. If the new value for the *control-variable* is greater than the *limit* (if the *increment* is positive) or less than the *limit* (if the *increment* is negative), the computer leaves the loop and continues with the program statement following the NEXT statement. The value of the *control-variable* is not changed when the computer leaves the FOR-NEXT loop.

You control the number of times the FOR-NEXT loop is performed by the values you assign in the FOR-TO-STEP statement. The *limit,* and, optionally, the STEP *increment* are numeric-expressions that are evaluated once during a loop performance (when the FOR-TO-STEP statement is encountered) and remain in effect until the loop is finished. Any change made to these values while a loop is in progress has no effect on the number of times the loop is performed. If the value of the *increment* is zero, the computer displays the error message "BAD VALUE IN xx" and the program stops running.

Examples:

```
>NEW

>100 REM COMPUTING SIMPLE
 INTEREST FOR 10 YEARS
>110 INPUT "PRINCIPLE? ":P
>120 INPUT "RATE? ":R
>130 FOR YEARS=1 TO 10
>140 P=P+(P*R)
>150 NEXT YEARS
>160 P=INT(P*100+.5)/100
>170 PRINT P
>180 END
>RUN
 PRINCIPLE? 100
 RATE? .0775
  210.95

 ** DONE **
```

```
>NEW

>100 REM EXAMPLE OF
 FRACTIONAL INCREMENT
>110 FOR X=.1 TO 1 STEP .2
>120 PRINT X;
>130 NEXT X
>140 PRINT :X
>150 END
>RUN
  .1  .3  .5  .7  .9
  1.1

 ** DONE **
```

```
>NEW

>100 L=5
>110 FOR I=1 TO L
>120 L=20
>130 PRINT L;I
>140 NEXT I
>150 END
>RUN
  20  1
  20  2
  20  3
  20  4
  20  5

 ** DONE **
```

FOR-TO-STEP

After you enter a RUN command, but before your program is performed, the computer checks to see that you have the same number of FOR-TO-STEP and NEXT statements. If you do not have the same number, the message "FOR-NEXT ERROR" is displayed and the program is not run.

Examples:

If you change the value of the *control-variable* while the loop is performed, the number of times the loop is repeated is affected.

```
>NEW

>100 FOR I=1 TO 10
>110 I=I+1
>120 PRINT I
>130 NEXT I
>140 PRINT I
>150 END
>RUN
  2
  4
  6
  8
  10
  11

 ** DONE **
```

In TI BASIC the expressions for *initial-value, limit,* and *increment* are evaluated before the *initial-value* is assigned to the *control-variable.* Thus, in the program on the right, in line 110 the value 5 is assigned to the *limit* before assigning a value to I as the *control-variable.* The loop is repeated 5 times, not just once.

```
>NEW

>100 I=5
>110 FOR I=1 TO I
>120 PRINT I;
>130 NEXT I
>140 END
>RUN
  1  2  3  4  5
 ** DONE **
```

The sign of the *control-variable* can change during the performance of a FOR-NEXT loop.

```
>NEW

>100 FOR I=2 TO -3 STEP -1
>110 PRINT I;
>120 NEXT I
>130 END
>RUN
  2  1  0 -1 -2 -3
 ** DONE **
```

When performing the FOR statement, the computer checks that the *limit* exceeds the *initial-value* before it does the loop. The *initial-value* in the FOR statement does *not* have to be 1. The computer can begin counting with whatever numeric value you wish. However, if the *initial-value* is greater than the *limit* and the *increment* is positive, the loop will not be performed at all. The computer will continue on to the statement following the loop. Similarly, if the *increment* is negative and you assign an *initial-value* less than the *limit,* the loop will not be performed.

```
>NEW

>100 REM INITIAL VALUE TOO
 GREAT
>110 FOR I=6 TO 5
>120 PRINT I
>130 NEXT I
>140 END
>RUN

 ** DONE **
```

FOR-TO-STEP

FOR-NEXT loops may be "nested"; that is, one FOR-NEXT loop may be contained wholly within another. You must use caution, however, to observe the following conventions:

- Each FOR-TO-STEP statement must be paired with a NEXT statement.
- Different *control-variables* must be used for each nested FOR-NEXT loop.
- If a FOR-NEXT loop contains any portion of another FOR-NEXT loop, it must contain *all* of the second FOR-NEXT loop.

Otherwise, the computer will stop running your program and print the error message "CAN'T DO THAT IN xx" if a FOR-NEXT loop overlaps another.

You may branch out of a FOR-NEXT loop using GOTO and IF-THEN-ELSE statements, but you may not branch into a FOR-NEXT loop using these statements. You may use GOSUB statements to leave a FOR-NEXT loop and return. Be sure you do not use the same *control-variable* for any FOR-NEXT loops you may have in your subroutines.

Examples:

```
>NEW

>100 REM FIND THE LOWEST
 THREE DIGIT NUMBER EQUAL TO
 THE SUM OF THE CUBES OF ITS
 DIGITS
>110 FOR HUND=1 TO 9
>120 FOR TENS=0 TO 9
>130 FOR UNITS=0 TO 9
>140 SUM=100*HUND+10*TENS+UNI
 TS
>150 IF SUM<>HUND^3+TENS^3+UN
 ITS^3 THEN 180
>160 PRINT SUM
>170 GOTO 210
>180 NEXT UNITS
>190 NEXT TENS
>200 NEXT HUND
>210 END
>RUN
  153

 ** DONE **
```

```
>NEW

>100 FOR I=1 TO 3
>110 PRINT I
>120 GOSUB 140
>130 NEXT I
>140 FOR I=1 TO 5
>150 PRINT I;
>160 NEXT I
>170 RETURN
>180 END
>RUN
  1
  1  2  3  4  5
 * CAN'T DO THAT IN 130
```

NEXT

NEXT *control-variable*

The NEXT statement is always paired with the FOR-TO-STEP statement for construction of a loop. The *control-variable* is the same one that appears in the corresponding FOR-TO-STEP statement.

The NEXT statement actually controls whether the computer will repeat the loop or exit to the program line following the NEXT statement.

When the computer encounters the NEXT statement, it adds the previously evaluated *increment* in the STEP clause to the *control-variable.* It then tests the *control-variable* to see if it exceeds the previously evaluated *limit* specified in the FOR-TO-STEP statement. If the *control-variable* does not exceed the *limit,* the loop is repeated.

Examples:

```
>NEW

>100 REM COUNTING FROM 1 TO
 10
>110 FOR X=1 TO 10
>120 PRINT X;
>130 NEXT X
>140 END
>RUN
  1  2  3  4  5  6  7  8  9
  10
 ** DONE **
```

```
>NEW

>100 REM ROCKET COUNTDOWN
>110 CALL CLEAR
>120 FOR I=10 TO 1 STEP -1
>130 PRINT I
>140 FOR DELAY=1 TO 200
>150 NEXT DELAY
>160 CALL CLEAR
>170 NEXT I
>180 PRINT "BLAST OFF!"
>190 REM CHANGE SCREEN COLOR
>200 FOR COLOR=2 TO 16 STEP 2
>210 CALL SCREEN(COLOR)
>220 FOR DELAY=1 TO 100
>230 NEXT DELAY
>240 NEXT COLOR
>250 END
>RUN

--computer will flash countdow

 BLAST OFF!

--screen will change color
  8 times

 ** DONE **
```

Input-Output Statements

Introduction

INPUT-OUTPUT statements allow you to transfer data in and out of your program. This section describes these statements (PRINT, DISPLAY, INPUT, READ, DATA, RESTORE) as they are used with your TI Home Computer keyboard and screen.

Data can be input to your program from three types of sources:

- from the keyboard – using the INPUT statement
- internally from the program itself – using the READ, DATA, and RESTORE statements
- from files stored on accessory devices – using the INPUT statement

Data can go to two types of output devices:

- the screen – using the PRINT or DISPLAY statements
- files stored on accessory devices – using the PRINT statement

There are two other sections in this Reference Guide which describe additional input-output capabilities of the TI Home Computer. The File Processing Section will help you construct the statements used with accessory devices. And, since your TI Home Computer is enhanced by graphics, color, and sound, many built-in subprograms also serve an input-output function. The Color Graphics and Sound Section will show you how to use these features.

INPUT

INPUT [*input-prompt:*] *variable-list*

(For information on the use of the INPUT statement with a file, see the section on File Processing beginning on page 144.)

This form of the INPUT statement is used when entering data via the keyboard. The INPUT statement causes the program to pause until valid data is entered from the keyboard. Although the computer usually accepts up to one input line (4 lines on your screen) for each INPUT statement, a long list of values may be rejected by the computer. If you receive the message "LINE TOO LONG" after entering an input line, you will need to divide the lengthy INPUT statement into at least two separate statements.

Examples:

Entering the Input Statement

The *input-prompt* is a string expression that indicates on the screen the values you should enter at that time. Including an *input-prompt* in the INPUT statement is optional. When the computer performs an INPUT statement that does not have an *input-prompt*, it displays a question mark (?) followed by a space and waits for you to enter your data.

```
>NEW

>100 INPUT B
>110 PRINT B
>120 END
>RUN
 ? 25
  25

 ** DONE **
```

If you use an *input-prompt*, the string expression must be followed by a colon. When the computer performs this type of INPUT statement, it will display the *input-prompt* message on the screen and wait for you to enter your data.

```
>NEW

>100 INPUT "COST OF CAR?":B
>110 A$="TAX?"
>120 INPUT A$:C
>130 INPUT "SALES "&A$:X
>140 PRINT B;C;X
>150 END
 RUN
 COST OF CAR?5500
 TAX?500
 SALES TAX?500
  5500  500  500

 ** DONE **
```

The *variable-list* contains those variables which are assigned values when the INPUT statement is performed. Variable names in the *variable-list* are separated by commas and may be numeric and/or string variables.

```
>NEW

>100 INPUT A,B$,C,D
>110 PRINT A:B$:C:D
>120 END
 RUN
 ? 10,HELLO,25,3.2
  10
 HELLO
  25
  3.2

 ** DONE **
```

INPUT

Responding to an Input Statement

When an INPUT statement is performed, the values corresponding to the variables must be entered in the same order as they are listed in the INPUT statement. When you enter the values, they must all be entered in one input line (up to 4 screen lines) with the values separated by commas. When inputting string values, you may enclose the string in quotes. However, if the string you wish to input contains a comma, a leading quote mark, leading spaces, or trailing spaces, it *must* be enclosed in quotes.

Examples:

```
>NEW

>100 INPUT A$
>110 PRINT A$::
>120 INPUT B$
>130 PRINT B$::
>140 INPUT C$
>150 PRINT C$::
>160 INPUT D$
>170 X=500
>180 PRINT D$;X::
>190 INPUT E$
>200 PRINT E$
>210 END
 RUN
 ? "JONES, MARY"
 JONES, MARY

 ? """HELLO THERE"""
 "HELLO THERE"

 ? "JAMES B. SMITH, JR."
 JAMES B. SMITH, JR.

 ? "SELLING PRICE IS "
 SELLING PRICE IS  500

 ? TEXAS
 TEXAS

 ** DONE **
```

Variables are assigned values from left to right in the *variable-list*. Thus, subscript expressions in the *variable-list* are not evaluated until variables to the left have been assigned values.

```
>NEW

>100 INPUT I,A(I)
>110 PRINT I:A(3)
>120 END
 RUN
 ? 3,7
   3
   7

 ** DONE **
```

INPUT

When input information is entered, it is validated by the computer. If the input data is invalid, the message "WARNING: INPUT ERROR, TRY AGAIN" appears on the screen and you must reenter the line. Here are some causes of this message:

- if you try to enter input data that contains more or fewer values than requested by the INPUT statement.
- if you try to enter a string constant when a number is required. (Remember, a number is a valid string, so you may enter a number when a string constant is required.)

If a number is input that causes an overflow (see page 37), the message "WARNING: NUMBER TOO BIG, TRY AGAIN" appears on the screen and you must reenter the line. If a number is input that causes an underflow (see page 37), the value is replaced by zero. No warning message is given.

Examples:

```
>NEW

>100 INPUT A,B$
>110 PRINT A;B$
>120 END
>RUN
 ? 12,HI,3

 * WARNING:
   INPUT ERROR IN 100
 TRY AGAIN: HI,3

 * WARNING:
   INPUT ERROR IN 100
 TRY AGAIN: 23,HI
  23 HI

 ** DONE **
```

```
>NEW

>100 INPUT A
>110 PRINT A
>120 END
>RUN
 ? 23E139

 * WARNING:
   NUMBER TOO BIG IN 100
 TRY AGAIN: 23E-139
 0

 ** DONE **
```

READ

READ *variable-list*

The READ statement allows you to read data stored inside your program in DATA statements (see page 91). *The variable-list* specifies those variables that are to have values assigned. Variable names in the *variable-list* are separated by commas. The *variable-list* may include numeric variables and/or string variables.

Examples:

```
>NEW

>100 FOR I=1 TO 3
>110 READ X,Y
>120 PRINT X;Y
>130 NEXT I
>140 DATA 22,15,36,52,48,96.5

>150 END
>RUN
  22  15
  36  52
  48  96.5

 ** DONE **
```

The computer reads each DATA statement sequentially from left to right and assigns values to the variables in the *variable-list* from left to right. Subscript expressions in the *variable-list* are not evaluated until variables to the left have been assigned.

```
>NEW

>100 READ I,A(I)
>110 DATA 2,35
>120 PRINT A(2)
>130 END
>RUN
  35

 ** DONE **
```

DATA statements are normally read in line-number order. Each time a READ statement is performed, values for the variables in the *variable-list* are assigned sequentially, using all the items in the data-list of the current DATA statement before moving to the next DATA statement. You can override this sequencing, however, by using the RESTORE statement (see page 92).

By following the program on the right, you can see how the READ, DATA, and RESTORE statements interact. In line 120 the computer begins assigning values to A and B from the DATA statement with the lowest line number, line 180. The first READ, therefore, assigns A=2 and B=4. The next performance of the READ statement still takes data from line 180 and assigns A=6, B=8. The third READ assigns the last item in line 180 to the variable A and the first item in line 190 to the variable B, so A=10, B=12. The fourth READ, the last in the J-loop, continues to get data from line 190, so A=14, B=16. Before going through the I-loop again, however, note that the computer encounters a RESTORE statement in line 160 which directs it to get data from the beginning of line 190 for the next READ statement. The computer then completes the program by reading the data from line 190 and then from line 200.

```
>NEW

>100 FOR I=1 TO 2
>110 FOR J=1 TO 4
>120 READ A,B
>130 PRINT A;B;
>140 NEXT J
>150 PRINT
>160 RESTORE 190
>170 NEXT I
>180 DATA 2,4,6,8,10
>190 DATA 12,14,16,18
>200 DATA 20,22,24,26
>210 END
>RUN
  2  4  6  8  10  12  14  16
  12  14  16  18  20  22  24
  26

 ** DONE **
```

READ

When data is read from a DATA statement, the type of data in the data-list and the type of variables to which the values are assigned must correspond. If you try to assign a string value to a numeric variable, the message "DATA ERROR IN xx" (xx is the line number of the READ statement where the error occurs) appears on the screen and the program stops running. Remember that a number is a valid string so numbers may be assigned to either string or numeric variables.

Examples:

```
>NEW

>100 READ A,B
>110 DATA 12,HELLO
>120 PRINT A;B
>130 END
>RUN

 * DATA ERROR IN 100

>□
```

When a READ statement is performed, if there are more names in the *variable-list* than values remaining in DATA statements, a "DATA ERROR" message is displayed on the screen and the program stops running. If a numeric constant is read which causes an underflow (see page 37), its value is replaced by zero — no warning is given — and the program continues running normally. If a numeric constant is read which causes an overflow (see page 37), its value is replaced by the appropriate computer limit (see page 37), the message "WARNING: NUMBER TOO BIG" is displayed on the screen, and the program continues.

```
>NEW

>100 READ A,B
>110 DATA 12E-135
>120 DATA 36E142
>130 PRINT :A:B
>140 READ C
>150 END
>RUN

 * WARNING:
   NUMBER TOO BIG IN 100

 0
 9.99999E+**

 * DATA ERROR IN 140

>□
```

DATA

DATA *data-list*

Examples:

The DATA statement allows you to store data inside your program. Data in the *data-lists* are obtained via READ statements when the program is run. The *data-list* contains the values to be assigned to the variables specified in the variable-list of a READ statement (see page 89). Items in the *data-list* are separated by commas. When a program reaches a DATA statement, it proceeds to the next statement with no other effect.

DATA statements may appear anywhere in a program, but the order in which they appear is important. Data from the *data-lists* are read sequentially, beginning with the first item in the first DATA statement. If your program includes more than one DATA statement, the DATA statements are read in ascending line-number order unless otherwise specified by a RESTORE statement (see page 92). Thus, the order in which the data appears within the *data-list* and the order of the DATA statements within the program normally determines in which order the data is read.

```
>NEW

>100 FOR I=1 TO 5
>110 READ A,B
>120 PRINT A;B
>130 NEXT I
>140 DATA 2,4,6,7,8
>150 DATA 1,2,3,4,5
>160 END
>RUN
  2  4
  6  7
  8  1
  2  3
  4  5

 ** DONE **
```

Data in the *data-list* must correspond to the type of the variable to which it is assigned. Thus, if a numeric variable is specified in the READ statement, a numeric constant must be in the corresponding place in the DATA statement. Similarly, if a string variable is specified, a string constant must be in the corresponding place in the DATA statement. Remember that a number is a valid string, so you may have a number in the corresponding place in the DATA statement when a string constant is required.

When using string constants in a DATA statement, you may enclose the string in quotes. However, if the string you include contains a comma, a leading quote mark, leading spaces, or trailing spaces, it *must* be enclosed in quotes.

```
>NEW

>100 READ A$,B$,C,D
>110 PRINT A$:B$:C:D
>120 DATA HELLO,"JONES, MARY"
 ,28,3.1416
>130 END
>RUN
 HELLO
 JONES, MARY
  28
  3.1416

 ** DONE **
```

If the list of string constants in the DATA statement contains adjacent commas, the computer assumes you want to enter a null string (a string with no characters). In the example on the right, the DATA statement in line 110 contains two adjacent commas. Thus, a null string is assigned to B$, as you can see when the program is run.

```
>NEW

>100 READ A$,B$,C
>110 DATA HI,,2
>120 PRINT "A$ IS ";A$
>130 PRINT "B$ IS ";B$
>140 PRINT "C IS ";C
>150 END
 RUN
 A$ IS HI
 B$ IS
 C IS  2

 ** DONE **
```

RESTORE

RESTORE [*line-number*]

(See page 162 for using RESTORE for file processing.)

This form of the RESTORE statement tells your program which DATA statement to use with the next READ statement.

When RESTORE is used with no *line-number* and the next READ statement is performed, values will be assigned beginning with the first DATA statement in the program.

When RESTORE is followed by the *line-number* of a DATA statement and the next READ statement is performed, values will be assigned beginning with the first data-item in the DATA statement specified by the *line-number*.

If the *line-number* specified in a RESTORE statement is not a DATA statement or is not a program line number, then the next READ statement performed will start at the first DATA statement whose line number is greater than the one specified. If there is no DATA statement with a line number greater than or equal to the one specified, then the next READ statement performed will cause an out-of-data condition and a "DATA ERROR" message will be displayed. If the *line-number* specified is greater than the highest line number in the program, the program will stop running and the message "DATA ERROR IN xx" will be displayed.

Examples:

```
>NEW

>100 FOR I=1 TO 2
>110 FOR J=1 TO 4
>120 READ A
>130 PRINT A;
>140 NEXT J
>150 RESTORE 180
>160 NEXT I
>170 DATA 12,33,41,26,42,50
>180 DATA 10,20,30,40,50
>190 END
>RUN
  12  33  41  26  10  20  30
  40
 ** DONE **
```

```
>NEW

>100 FOR I=1 TO 5
>110 READ X
>120 RESTORE
>130 PRINT X;
>140 NEXT I
>150 DATA 10,20,30
>160 END
>RUN
  10  10  10  10  10
 ** DONE **
```

```
>NEW

>100 READ A,B
>110 RESTORE 130
>120 PRINT A;B
>130 READ C,D
>140 PRINT C;D
>150 DATA 26.9,34.67
>160 END
>RUN
  26.9  34.67
  26.9  34.67

 ** DONE **

>110 RESTORE 145
>RUN
  26.9  34.67
  26.9  34.67

 ** DONE **

>110 RESTORE 155
>RUN
  26.9  34.67

 * DATA ERROR IN 130

>□
```

PRINT

PRINT *[print-list]*

(For information on the PRINT statement as used with files, see pages 157 to 161.)

The PRINT statement lets you print numbers and strings on the screen. The *print-list* consists of

- *print-items* – numeric expressions and string expressions which print on the screen and *tab-functions* which control print positioning (similar to the TAB key on the typewriter).
- *print-separators* – the punctuation between *print-items* (commas, colons, and semicolons) which serves as indicators for positioning data on the print-line.

When the computer performs a PRINT statement, the values of the expressions in the *print-list* are displayed on the screen in order from left to right, as specified by the *print-separators* and *tab-functions*.

Examples:

```
>NEW

>100 A=10
>110 B=20
>120 STRING$="TI COMPUTER"
>130 PRINT A;B:STRING$
>140 PRINT "HELLO, FRIEND"
>150 END
>RUN
  10  20
 TI COMPUTER
 HELLO, FRIEND

 ** DONE **
```

Printing Strings

String expressions in the *print-list* are evaluated to produce a string result. There are no blank spaces inserted before or after a string. If you wish to print a blank space before or after a string you can include it in the string or insert it separately with quotes.

```
>NEW

>100 N$="JOAN"
>110 M$="HI"
>120 PRINT M$;N$
>130 PRINT M$&" "&N$
>140 PRINT "HELLO ";N$
>150 END
>RUN
 HIJOAN
 HI JOAN
 HELLO JOAN

 ** DONE **
```

Printing Numbers

Numeric expressions in the *print-list* are evaluated to produce a numeric result to be printed. Positive numbers are printed with a leading space (instead of a plus sign) and negative numbers are printed with a leading minus sign. All numbers are printed with a trailing space.

```
>NEW

>100 LET A=10.2
>110 B=-30.5
>120 C=16.7
>130 PRINT A;B;C
>140 PRINT A+B
>150 END
>RUN
  10.2 -30.5  16.7
 -20.3

 ** DONE **
```

PRINT

The PRINT statement displays numbers in either *normal decimal form* or *scientific notation* (see page 37), according to these rules:

Examples:

1. All numbers with 10 or fewer digits are printed in *normal decimal form.*

```
>PRINT -10;7.1
 -10  7.1
```

2. Integer numbers with more than 10 digits are printed in *scientific notation.*

```
>PRINT 93427685127
  9.34277E+10
```

3. Non-integer numbers with more than 10 digits are printed in *scientific notation* only if they can be presented with more significant digits in *scientific notation* than in *normal decimal form.* If printed in *normal decimal form,* all digits beyond the tenth digit are omitted.

```
>PRINT 1E-10
  .0000000001

>PRINT 1.2E-10
  1.2E-10

>PRINT .000000000246
  2.46E-10
```

   If numbers are printed in *normal decimal form,* the following conventions are observed:

   - Integers are printed with no decimal point.

```
>PRINT 15;-3
  15 -3
```

   - Non-integers have the decimal point printed in its proper place. Trailing zeros in the fractional part are omitted. If the number has more than ten digits, the tenth digit is rounded.

```
>PRINT 3.350;-46.1
  3.35 -46.1

>PRINT 791.123456789
  791.1234568
```

   - Numbers with a value less than one are printed with *no* digits to the left of the decimal point.

```
>PRINT -12.7E-3;0.64
 -.0127  .64
```

If numbers are printed in *scientific notation,* the format is:

mantissa E exponent

and the following rules apply:

- The mantissa is printed with 6 or fewer digits and is always displayed with one digit to the left of the decimal point.

```
>PRINT .0000000001978531
  1.97853E-10

>PRINT -98.77E21
 -9.877E+22
```

- Trailing zeros are omitted in the fractional part of the mantissa.

```
>PRINT 736.400E10
  7.364E+12
```

- If there are more than five digits in the fractional part of the mantissa, the fifth digit is rounded.

```
>PRINT 12.36587E-15
  1.23659E-14
```

- The exponent is displayed with a plus or minus sign followed by a two-digit number.

```
>PRINT 1.25E-9;-43.6E12
  1.25E-09 -4.36E+13
```

- If you attempt to print a number with an exponent value larger than +99 or smaller than −99, the computer will print ** following the proper sign of the exponent.

```
>PRINT .76E126;81E-115
  7.6E+**  8.1E-**
```

PRINT

Print-Separators

Examples:

Each screen line used with the PRINT statement has 28 character positions numbered from left to right (1-28). Each line is divided into two 14-character print zones. By using the *print-separators* and the *tab-function*, you can control the position of the *print-items* displayed on the screen.

There are three types of *print-separators:* semicolons, colons, and commas. At least one *print-separator* must be placed between adjacent *print-items* in the *print-list*. Multiple *print-separators* may be used side by side and are evaluated from left to right.

```
>PRINT "A"::"B"
 A

 B
```

The *semicolon print-separator* causes adjacent *print-items* to print side by side with no extra spaces between the values. In the program on the right, the spaces after the numbers appear only because all numbers are printed with a trailing space regardless of the type of *print-separator* used.

```
>NEW

>100 A=-26
>110 B=-33
>120 C$="HELLO"
>130 D$="HOW ARE YOU?"
>140 PRINT A;B;C$;D$
>150 END
>RUN
 -26 -33 HELLOHOW ARE YOU?

 ** DONE **
```

The *colon print-separator* causes the next *print-item* to print at the beginning of the next line.

```
>NEW

>100 A=-26
>110 B$="HELLO"
>120 C$="HOW ARE YOU?"
>130 PRINT A:B$:C$
>140 END
>RUN
 -26
 HELLO
 HOW ARE YOU?

 ** DONE **
```

Print lines are divided into two zones. The first zone begins in column 1 and the second begins in column 15. When the computer evaluates a *comma print-separator*, the next *print-item* is printed at the beginning of the next zone. If it is already in the second print zone when a *comma print-separator* is evaluated, the next *print-item* is begun on the next line.

```
>NEW

>100 A$="ZONE 1"
>110 B$="ZONE 2"
>120 PRINT A$,B$
>130 PRINT A$:,B$,A$
>140 END
>RUN
 ZONE 1        ZONE 2
 ZONE 1
               ZONE 2
 ZONE 1

 ** DONE **
```

DISPLAY

DISPLAY [*print-list*]

The DISPLAY statement is identical to the PRINT statement when you use it to print items on the screen. The DISPLAY statement may not be used to write on any device except the screen. For a complete discussion of how to use this statement, follow the instructions for the PRINT statement beginning on page 93.

Examples:

```
>NEW

>100 A=35.6
>110 B$="HI!!"
>120 C=49.7
>130 PRINT B$:A;C
>140 DISPLAY B$:A;C
>150 END
>RUN
 HI!!
  35.6   49.7
 HI!!
  35.6   49.7

 ** DONE **
```

A *print-item* will not be split between two screen lines unless the *print-item* is a string with more than twenty-eight characters. In that case the string is always begun on a new line and is printed with twenty-eight characters per line until the entire string is printed. If a numeric *print-item* is such that the only character not able to fit on the current line is a trailing space, then the number will be printed on the current line. If the number itself will not fit on the current line, it is printed on the next line.

The *print-list* may end with a *print-separator*. If the *print-list* is not terminated by a *print-separator* (line 130), the computer considers the current line completed when all the characters produced from the *print-list* are printed. In this case the first *print-item* in the next PRINT statement (line 140) always begins on a new line.

If the *print-list* ends with a *print-separator* (line 140), then the *print-separator* is evaluated and the first *print-item* in the next PRINT statement (line 160) will start in the position indicated by the *print-separator*.

You may use a PRINT statement with no *print-list*. When such a PRINT statement is performed, the computer advances to the first character position of the next screen line. This has the effect of skipping a line if the preceding PRINT statement has no *print-separator* at the end.

Examples:

```
>NEW

>100 A=23767
>110 B=79856
>120 C=A+B
>130 D=B-A
>140 PRINT A;B;C;D
>150 PRINT "A=";A;"B=";B;"C="
 ;C;"D=";D
>160 END
>RUN
  23767  79856  103623  56089
 A= 23767 B= 79856 C= 103623
 D= 56089

 ** DONE **
```

```
>NEW

>100 A=23
>110 B=597
>120 PRINT A,
>130 PRINT B
>140 PRINT A;B;
>150 C=468
>160 PRINT C
>170 END
>RUN
  23           597
  23  597  468

 ** DONE **
```

```
>NEW

>100 A=20
>110 PRINT A
>120 PRINT
>130 B=15
>140 PRINT B
>150 END
>RUN
  20

  15

 ** DONE **
```

```
>NEW

>100 FOR J=1 TO 2
>110 FOR I=1 TO 3
>120 PRINT I;
>130 NEXT I
>140 PRINT
>150 NEXT J
>160 END
>RUN
  1  2  3
  1  2  3

 ** DONE **
```

CLEAR subprogram

CALL CLEAR

The CLEAR subprogram is used to clear (erase) the entire screen. When the CLEAR subprogram is called, the space character (code 32) is placed in all positions on the screen.

Examples:

```
>PRINT "HELLO THERE!"
 HELLO THERE!
>CALL CLEAR

 --screen clears
```

When the program on the right is run, the screen is cleared before the PRINT statements are performed.

```
>NEW

>100 CALL CLEAR
>110 PRINT "HELLO THERE!"
>120 PRINT "HOW ARE YOU?"
>130 END
>RUN

 --screen clears

 HELLO THERE!
 HOW ARE YOU?

 ** DONE **
```

If the space character (code 32) has been redefined by the CALL CHAR subprogram (see page 104), the screen will be filled with the new character when the CALL CLEAR is performed, rather than with spaces.

```
>NEW

>100 CALL CHAR(32,"0103070F1F
 3F7FFF")
>110 CALL CLEAR
>120 GOTO 120
>RUN

 --screen will be filled
   with ◢

  (Press SHIFT C to stop
   the program.)
```

Color Graphics and Sound

Introduction

A special set of subprograms has been built into the TI Home Computer to provide color graphics, sound, and other capabilities not usually found in BASIC.

Whenever you want to use one of these special subprograms, you *call* for it by name and supply a few specifications. The subprogram then takes over, performs its task, and provides you with such things as musical tones, screen colors, and special graphics characters. These features are particularly useful when you are programming simulations, graphs, patterns on the screen, or your own "computer music." All of the subprograms may be used in Command Mode as well as in programs.

The built-in subprograms can be grouped according to their function:

- INPUT subprograms – GCHAR, JOYST, KEY
- OUTPUT subprograms – CLEAR, HCHAR, VCHAR, SOUND, SCREEN
- INTERNAL subprograms – CHAR, COLOR (the results of these subprograms aren't evident until you use an OUTPUT operation to see the results on the screen).

The graphics subprograms feature a 24-row by 32-column screen display. The 28 print positions normally used in TI BASIC correspond to columns 3 through 30, inclusive, in the graphics subprograms. Because some display screens may not show the two leftmost and two rightmost characters, your graphics may be more satisfactory if you use columns 3 through 30 and ignore columns 1 and 2 on the left and 31 and 32 on the right. Experiment with different line lengths to determine how many positions show on your screen.

COLOR subprogram

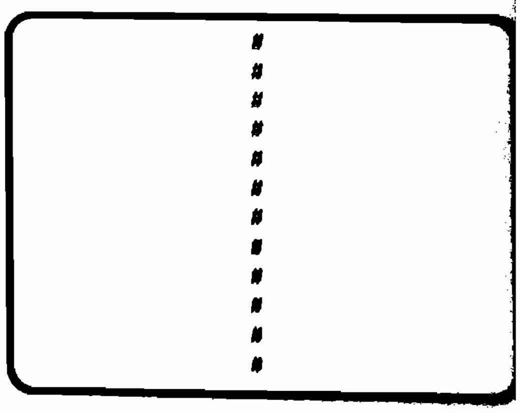
To use CALL COLOR you must also specify to which of sixteen character sets the character you are printing belongs. The list of ASCII character codes for the standard characters is given in the Appendix, page 163. The character will be displayed in the color specified when you use CALL HCHAR (page 108) or CALL VCHAR (page 111). The *character-set-numbers* are given below.

Set Number	*Character 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
9	96-103
10	104-111
11	112-119
12	120-127
13	128-135
14	136-143
15	144-151
16	152-159

Note that all 24 rows and 32 columns are filled with the space character until you place other characters in some of these positions. If you use character set 1 in the CALL COLOR statement, all space characters on the screen are changed to the *background-color* specified since the space character is contained in set 1. This change is demonstrated by the program on the right.

Examples:

```
>NEW

>100 CALL CLEAR
>110 CALL COLOR(1,16,14)
>120 CALL SCREEN(13)
>130 CALL VCHAR(1,15,35,24)
>140 GOTO 140
>RUN

 --screen clears

 --24 white #'s with
   a magenta background on a
   dark-green screen
```



```
#
#
#
#
#
#
#
#
#
#
#
#
```

```
 --Note that the screen color
   appears only at the top an
   bottom of the screen

(Press SHIFT C to stop
 the program)
```

COLOR subprogram

CALL COLOR *(character-set-number,foreground-color-code,background-color-code)*

The COLOR subprogram provides a powerful design capability by allowing you to specify screen character colors. (To change the screen color itself, see the SCREEN subprogram on page 103.) The *character-set-number, foreground-color-code,* and *background-color-code* are numeric expressions.

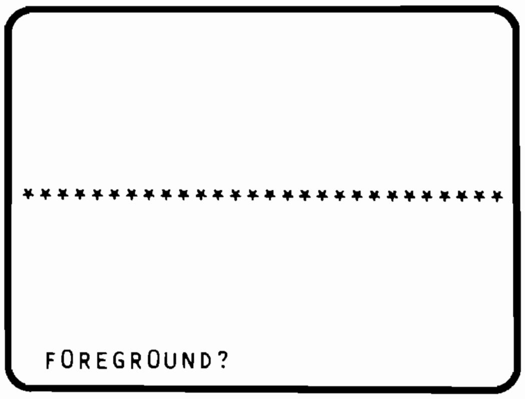
Each character displayed on your computer screen has two colors. The color of the dots that make up the character itself is called the *foreground color.* The color that occupies the rest of the character position on the screen is called the *background color.* Sixteen colors are available on the TI Home Computer so your entries for *foreground* and *background color* must have a value of 1 through 16. The color codes are given in the table below:

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

If transparent (code 1) is specified, the present screen color (see page 103) shows through when a character is displayed. Until a CALL COLOR is performed, the standard *foreground-color* is black (code 2) and the standard *background-color* is transparent (code 1) for all characters. When a breakpoint (see page 58) occurs, all characters are reset to the standard colors.

Examples:

```
>NEW

>100 CALL CLEAR
>110 INPUT "FOREGROUND?":F
>120 INPUT "BACKGROUND?":B
>130 CALL CLEAR
>140 CALL COLOR(2,F,B)
>150 CALL HCHAR(12,3,42,28)
>160 GO TO 110
>RUN

 --screen clears

 FOREGROUND?2
 BACKGROUND?14

 --screen clears

 (28 black asterisks with
  a magenta background)
```



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

 FOREGROUND?
```

```
  (Press SHIFT C to stop
   the program)
```

```
>NEW

>100 CALL CLEAR
>110 CALL SCREEN(12)
>120 CALL COLOR(2,1,7)
>130 CALL HCHAR(12,3,42,28)
>140 GOTO 140
>RUN

 --screen clears

 (transparent asterisks with
  a dark-red background on a
  light-yellow screen)

 ****************************

 (Press SHIFT C to stop
  the program)
```

CHAR subprogram

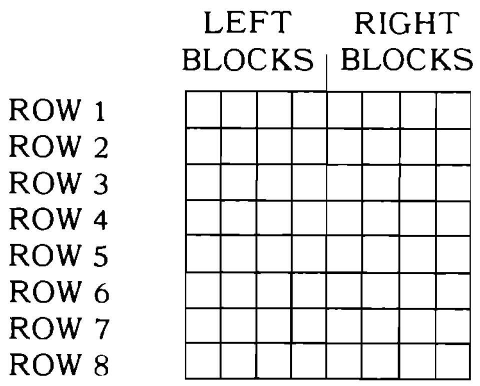
(Character definition)

CALL CHAR*(char-code,"pattern-identifier")*

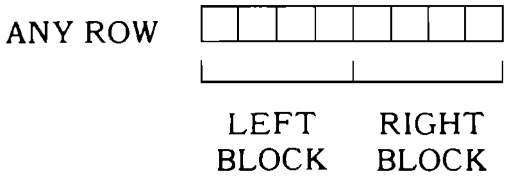
The CHAR subprogram allows you to define your own special graphics characters. You can redefine the standard set of characters (ASCII codes 32-95) and establish additional characters with codes 96-159.

The *char-code* specifies the code of the character you wish to define and must be a numeric expression with a value between 32 and 159, inclusive. If the character you are defining is in the range 96-159 and there is insufficient free memory to define the character, the program will terminate with a "MEMORY FULL" error.

The *pattern-identifier* is a 16-character string expression which specifies the pattern of the character you want to use in your program. This string expression is a coded representation of the 64 dots which make up a character position on the screen. These 64 dots comprise an 8-by-8 grid as shown below, greatly enlarged.



Each row is partitioned into two blocks of four dots each:



Examples:

```
>NEW

>100 CALL CLEAR
>110 CALL CHAR(33,"FFFFFFFFFF
 FFFFFF")
>120 CALL COLOR(1,9,6)
>130 CALL VCHAR(12,16,33)
>140 GOTO 140
>RUN

 --screen clears
            ■

 (Press SHIFT C to stop
  the program)
```

SCREEN subprogram

CALL SCREEN *(color-code)*

The SCREEN subprogram enhances the graphic capabilities of the TI Home Computer by allowing you to change the screen color. The standard screen color while a program is running is light green (*color-code* = 4).

The *color-code* is a numeric expression which, when evaluated, has a value of 1 through 16. The table of the sixteen available colors and their codes is given below.

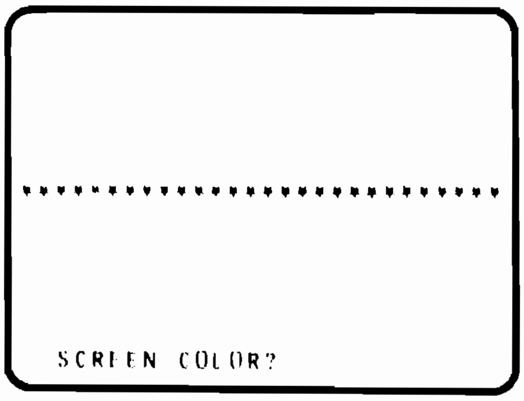
Color-code	*Color*
1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
9	Medium Red
10	Light Red
11	Dark Yellow
12	Light Yellow
13	Dark Green
14	Magenta
15	Gray
16	White

When the CALL SCREEN is performed, the entire screen background changes to the color specified by the *color-code*. All characters on the screen remain the same unless you have specified a transparent foreground or background color for them. In that case, the screen color "shows through" the transparent foreground or background.

The screen is set to cyan (code 8) when a program stops for a breakpoint (see page 58) or terminates. If you CONTINUE a program (see page 63) after a breakpoint, the screen will be reset to the standard color (light green).

Examples:

```
>NEW

>100 CALL CLEAR
>110 INPUT "SCREEN COLOR?":S
>120 INPUT "FOREGROUND?":F
>130 INPUT "BACKGROUND?":B
>140 CALL CLEAR
>150 CALL SCREEN(S)
>160 CALL COLOR(2,F,B)
>170 CALL HCHAR(12,5,42,28)
>180 GOTO 110
>RUN

 --screen clears

 SCREEN COLOR?7
 FOREGROUND?13
 BACKGROUND?16

 --screen clears

 --28 dark-green asterisks
   with a white background on
   a dark-red screen
```



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



  SCREEN COLOR?
```

```
 (Press SHIFT C to stop
  the program)
```

HCHAR subprogram

(Horizontal character repetition)

CALL HCHAR *(row-number, column-number, char-code* I*,number-of-repetitions*I*)*

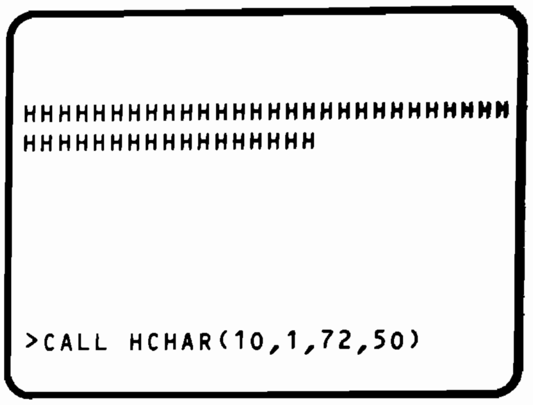
The HCHAR subprogram places a character anywhere on the screen and, optionally, repeats it horizontally. The *row-number* and *column-number* locate the starting position on the screen. The *row-number, column-number, char-code,* and *number-of-repetitions* are numeric expressions.

If the evaluation of any of the numeric expressions results in a non-integer value, the result is rounded to obtain an integer. The valid ranges are given below:

Value	*Range*
Row-number	1-24, inclusive
Column-number	1-32, inclusive
Char-code	0-32767, inclusive
Number-of-repetitions	0-32767, inclusive

Examples:

```
>CALL CLEAR

 --screen clears

>CALL HCHAR(10,1,72,50)
```



```
HHHHHHHHHHHHHHHHHHHHHHHHHHHH
HHHHHHHHHHHHHHHHH



>CALL HCHAR(10,1,72,50)
```

```
>NEW

>100 CALL CLEAR
>110 FOR S=2 TO 16
>120 CALL COLOR(S,S,S)
>130 NEXT S
>140 CHR=40
>150 FOR X=8 TO 22
>160 CALL VCHAR(4,X,CHR,15)
>170 CALL HCHAR(X-4,8,CHR,15)

>180 CHR=CHR+8
>190 NEXT X
>200 GOTO 140
>RUN

 --screen clears

 --makes a pattern on the
   screen using various COLORS

 (Press SHIFT C to stop
  the program)
```

CHAR subprogram

Each character in the string expression describes the pattern of dots in one block of a row. The rows are described from left to right and from top to bottom. That is, the first two characters in the string describe the pattern for row one of the dot-grid, the next two describe row two, and so on.

Characters are created by turning some dots "on" and leaving others "off." The space character (code 32) is a character with all the dots turned "off." Turning all the dots "on" produces a solid block (■).

All the standard characters are automatically set so that they turn "on" the appropriate dots. To create a new character, you must tell the computer what dots to turn on or leave off in each of the 16 blocks that contain the character. In the computer a binary code is used to specify what dots are on or off within a particular block. However, a "shorthand" method called hexadecimal, made up of numbers and letters, is used to control the on/off condition. The table that follows contains all the possible on/off conditions for the dots within a given block and the hexadecimal notation for each condition.

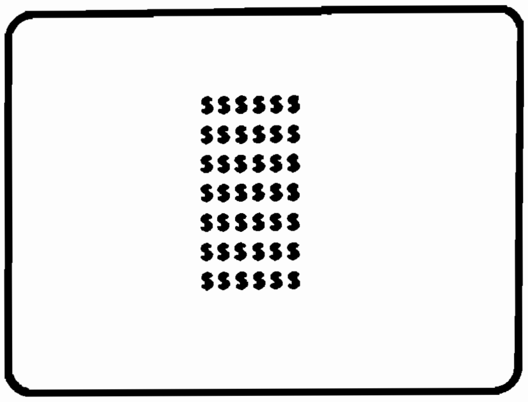
Blocks	*Binary Code (0 = Off; 1 = On)*	*Hexadecimal Code*
□□□□	0000	0
□□□■	0001	1
□□■□	0010	2
□□■■	0011	3
□■□□	0100	4
□■□■	0101	5
□■■□	0110	6
□■■■	0111	7
■□□□	1000	8
■□□■	1001	9
■□■□	1010	A
■□■■	1011	B
■■□□	1100	C
■■□■	1101	D
■■■□	1110	E
■■■■	1111	F

HCHAR subprogram

To repeat the specified character, enter a value for the *number-of-repetitions*. The computer will display the character beginning at the specified starting position and continue on the left side of the next line. If the bottom of the screen is reached, the display will continue on the top line of the screen. You should use 768 for *number-of-repetitions* to fill all 24 rows and 32 columns. Using a number larger than 768 will unnecessarily extend the time required to perform this statement.

Examples:

```
>NEW

>100 CALL CLEAR
>110 FOR I=9 TO 15
>120 CALL HCHAR(I,13,36,6)
>130 NEXT I
>140 GOTO 140
>RUN

 --screen clears
```



```
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
```

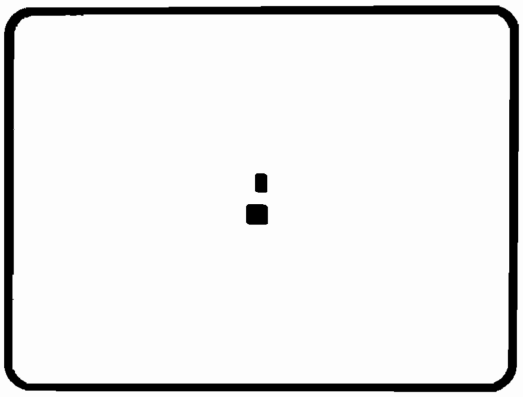
```
(Press SHIFT C to stop
    the program)
```

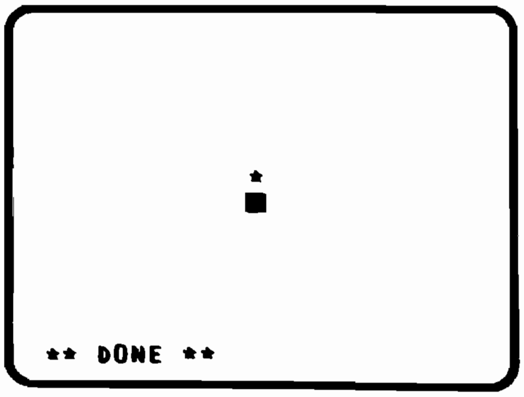
If a program stops for a breakpoint (see page 58), those characters redefining codes 32-95 are reset to their normal representation. Those with codes 96-159 are unchanged. When the program ends either normally or because of an error, all redefined characters are reset and any characters assigned to codes 96-159 are reset to be undefined.

Examples:

```
>NEW

>100 CALL CLEAR
>110 CALL CHAR(96,"FFFFFFFFFF
 FFFFFF")
>120 CALL CHAR(42,"0F0F0F0F0F
 0F0F0F")
>130 CALL HCHAR(12,17,42)
>140 CALL VCHAR(14,17,96)
>150 FOR DELAY=1 TO 350
>160 NEXT DELAY
>170 END
>RUN
```

 --screen clears





```
** DONE **
```

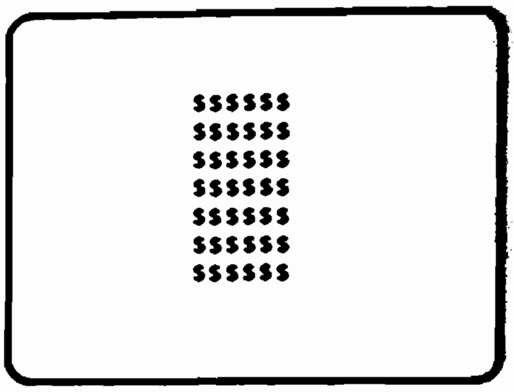
```
>CALL HCHAR(24,5,42)
   *
```

HCHAR subprogram

To repeat the specified character, enter a value for the *number-of-repetitions.* The computer will display the character beginning at the specified starting position and continue on the left side of the next line. If the bottom of the screen is reached, the display will continue on the top line of the screen. You should use 768 for *number-of-repetitions* to fill all 24 rows and 32 columns. Using a number larger than 768 will unnecessarily extend the time required to perform this statement.

Examples:

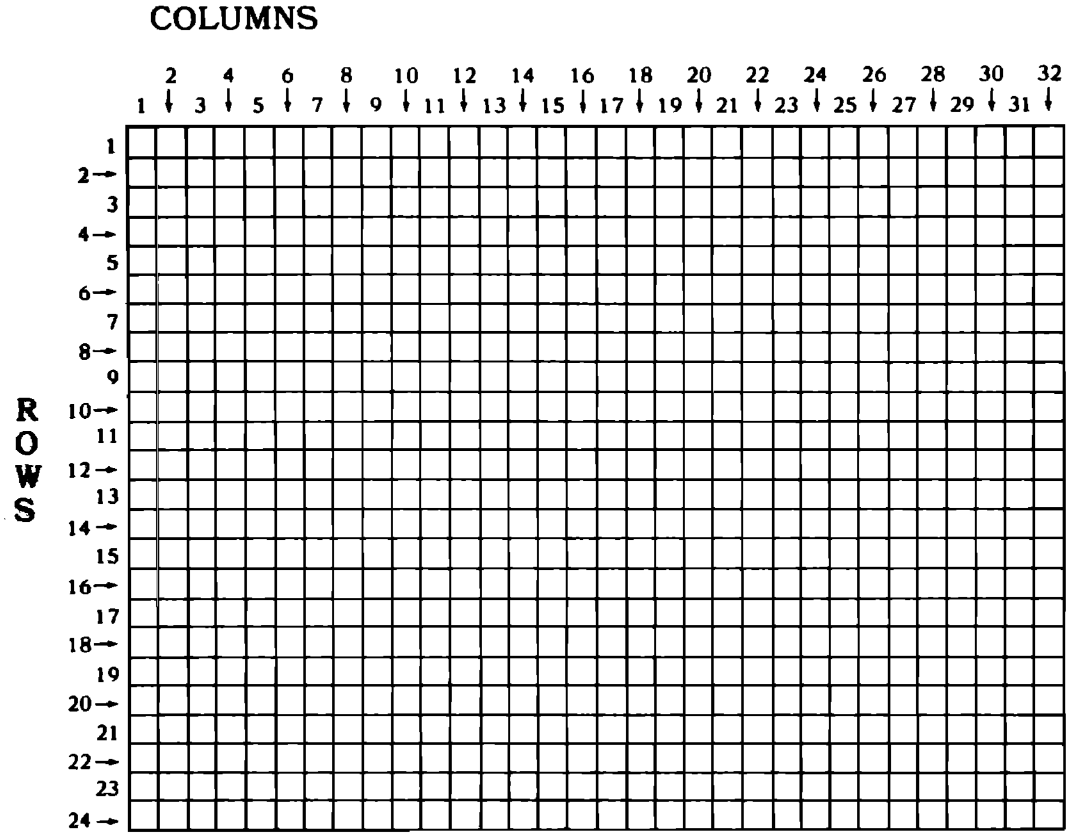
```
>NEW

>100 CALL CLEAR
>110 FOR I=9 TO 15
>120 CALL HCHAR(I,13,36,6)
>130 NEXT I
>140 GOTO 140
>RUN

 --screen clears
```



```
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
$$$$$$
```

```
(Press SHIFT C to stop
    the program)
```

A value of 1 for *row-number* indicates the top of the screen. A value of 1 for *column-number* indicates the left side of the screen. The screen can be thought of as a "grid" as shown here.



Because columns 1, 2, 31, and 32 may not show on your screen, you may want to use only *column-numbers* 3 through 30.

Although you may specify a value as large as 32767 for *char-code*, the computer will convert the value specified to a range of 0 through 255. Character codes 32 through 95 are defined as the standard ASCII character codes (see Appendix, page 163). Character codes 96 through 159 may be defined using the CHAR subprogram (see page 104). If you specify an undefined character for *char-code*, you will get whatever is in memory at the time the HCHAR subprogram is called.

```
>CALL HCHAR(24,14,29752)

            8

>CALL HCHAR(24,14,35)

            #

>CALL HCHAR(24,14,99)

 --displayed character depends
   on what is in memory now
```

SOUND subprogram

CALL SOUND(*duration,frequency1,volume1[,frequency2,volume2][,frequency3,volume3][,frequency4,volume4]*)

The SOUND subprogram tells the computer to produce tones of different frequencies. The values you include control three aspects of the tone:

- *duration* — how long the tone lasts.
- *frequency* — what tone actually plays.
- *volume* — how loud the tone is.

The *duration, frequency,* and *volume* are numeric expressions. If the evaluation of any of the numeric expressions results in a non-integer value, the result is rounded to obtain an integer. The valid ranges for each of these are given in the table and discussed further below.

Value	*Range*
duration	1 to 4250, inclusive –1 to –4250, inclusive
frequency	(Tone) 110 to 44733, inclusive (Noise) –1 to –8, inclusive
volume	0 (loudest) to 30 (quietest), inclusive

Duration

The *duration* you specify is measured in milliseconds. One second is equal to 1000 milliseconds. Thus, the duration ranges from .001 to 4.25 seconds. (The actual duration may vary as much as 1/60th of a second.) The *duration* you specify applies to each sound generated by a particular CALL SOUND statement.

In a program, the computer continues performing program statements while a sound is being played. When you call the SOUND subprogram, the computer will wait until the previous sound has been completed before performing the new CALL SOUND statement unless a negative *duration* is specified. If you specify a negative *duration* in the new CALL SOUND statement, the previous sound is stopped and the new one is begun immediately.

Examples:

```
>CALL SOUND(100,294,2)

 --plays a single tone
```

```
>NEW

>100 TONE=110
>110 FOR COUNT=1 TO 10
>120 CALL SOUND(-500,TONE,1)
>130 TONE=TONE+110
>140 NEXT COUNT
>150 END
>RUN

 -- plays ten tones quickly

 ** DONE **

>120 CALL SOUND(+500,TONE,1)
>RUN

 --plays ten tones slowly

 ** DONE **
```

VCHAR subprogram

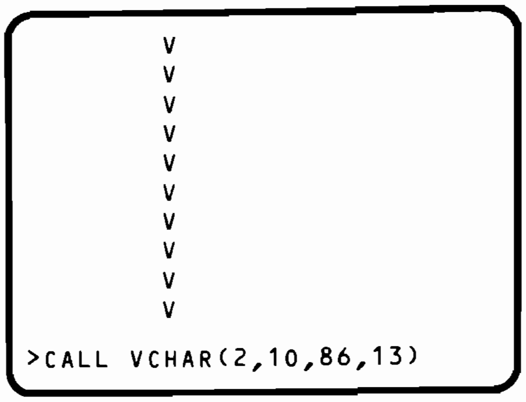
(Vertical character repetition)

CALL VCHAR *(row-number, column-number, char-code [,number-of-repetitions])*

The VCHAR subprogram performs very much like the HCHAR subprogram except that it repeats characters vertically rather than horizontally. The computer will display the character beginning at the specified position and continuing down the screen. If the bottom of the screen is reached, the display will continue at the top of the next column to the right. If the right edge of the screen is reached, the display will continue at the left edge. See the HCHAR subprogram on page 108 for more details.

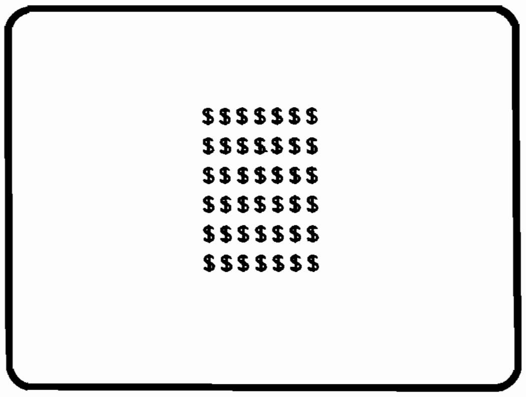
Examples:

```
>CALL CLEAR

 --screen clears

>CALL VCHAR(2,10,86,13)
```



```
         V
         V
         V
         V
         V
         V
         V
         V
         V
         V

>CALL VCHAR(2,10,86,13)
```

```
>NEW

>100 CALL CLEAR
>110 FOR I=13 TO 18
>120 CALL VCHAR(9,I,36,6)
>130 NEXT I
>140 GOTO 140
>RUN

 -- screen clears
```



```
            $$$$$$$
            $$$$$$$
            $$$$$$$
            $$$$$$$
            $$$$$$$
            $$$$$$$
```

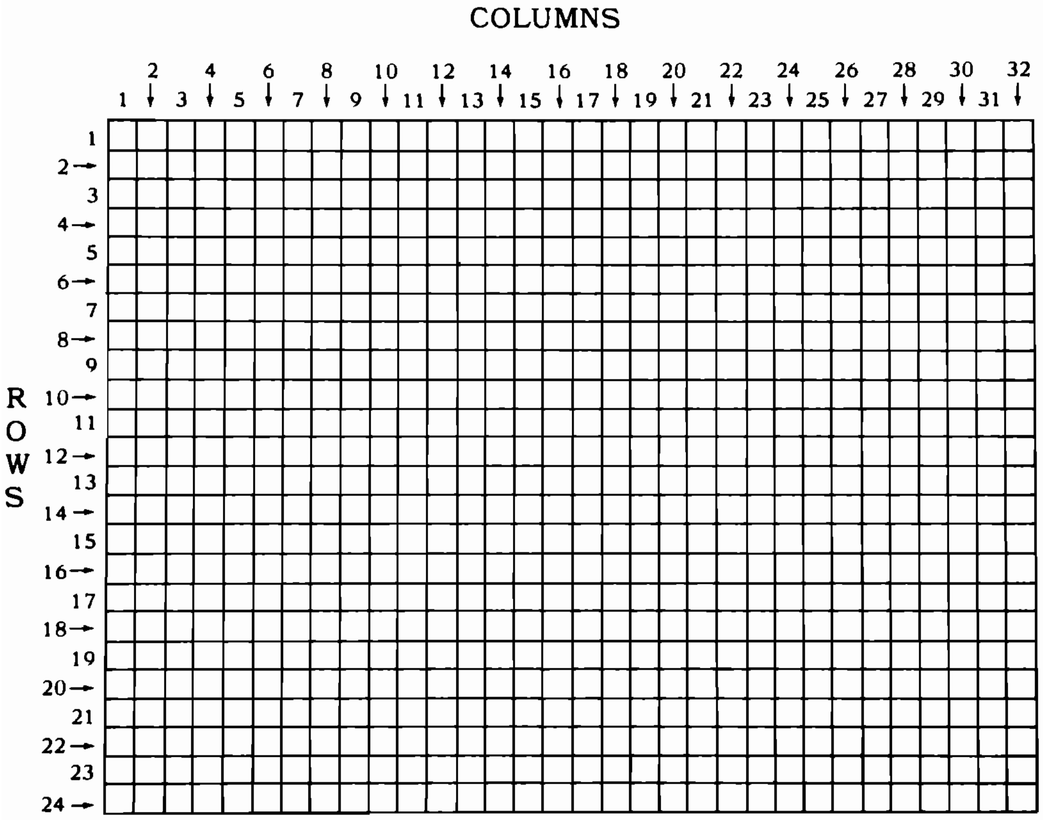
```
(Press SHIFT C to stop
   the program)
```

GCHAR subprogram
(Get character)

CALL GCHAR *(row-number,column-number,numeric-variable)*

The GCHAR subprogram allows you to read a character from anywhere on the display screen. The position of the character you want is described by *row-number* and *column-number*. The computer will put the ASCII numeric code (see Appendix, page 163) of the requested character into the *numeric-variable* you specify in the CALL GCHAR statement.

The *row-number* and *column-number* are numeric expressions. If the evaluation of the numeric expressions results in a non-integer value, the result is rounded to obtain an integer. A value of 1 for *row-number* indicates the top of the screen. A value of 1 for *column-number* specifies the left side of the screen. The screen can be thought of as a "grid" as shown here.



Examples:

```
>NEW

>100 CALL CLEAR
>110 CALL HCHAR(1,1,36,768)
>120 CALL GCHAR(5,10,X)
>130 CALL CLEAR
>140 PRINT X
>150 END
>RUN

  --screen clears

  --screen fills with $$$
    (code 36)

  --screen clears

   36

  ** DONE **
```

Frequency

The *frequency* you specify may be either a tone or a noise. The tones, measured in Hertz (one cycle per second, |Hz|), can be specified from a low-pitch of 110 Hz to a high pitch of 44733 Hz, well above human hearing limits. (The actual frequency produced may vary from zero to ten percent depending on the *frequency.)* The frequencies for some common musical notes are given in the Appendix, page 167.

If a negative value for *frequency* is specified, a noise, rather than a tone, is produced. The noise is either "white noise" or "periodic noise." The noise associated with each value is given in the table below. Since it is difficult to describe the difference between noises, you can try out the different noises yourself to become familiar with each one.

Noise Characteristics

Frequency Value	Characteristic
−1	"Periodic Noise" Type 1
−2	"Periodic Noise" Type 2
−3	"Periodic Noise" Type 3
−4	"Periodic Noise" that varies with the frequency of the third tone specified
−5	"White Noise" Type 1
−6	"White Noise" Type 2
−7	"White Noise" Type 3
−8	"White Noise" that varies with the frequency of the third tone specified

A maximum of three tones and one noise can be activated simultaneously. For each tone or noise specified, its volume must be indicated immediately following the tone or noise.

Examples:

```
>CALL SOUND(1000,440,2)

 --plays a single tone

>CALL SOUND(500,-1,2)

 --plays a single noise


>NEW

>100 FOR NOISE=-1 TO -8 STEP
 -1
>110 CALL SOUND(1000,NOISE,2)
>120 NEXT NOISE
>130 END
>RUN

 --all 8 different noises
     are generated

 ** DONE **


>CALL SOUND(2000,-3,5)

 --plays a single noise
```

```
>CALL SOUND(2500,440,2,659,5,
 880,10,-6,15)

 --plays 3 tones and 1 noise

>DUR=2500
>VOL=2
>C=262
>E=330
>G=392
>CALL SOUND(DUR,C,VOL,E,VOL,G
 ,VOL)

  --produces a C-major chord
```

JOYST Subprogram

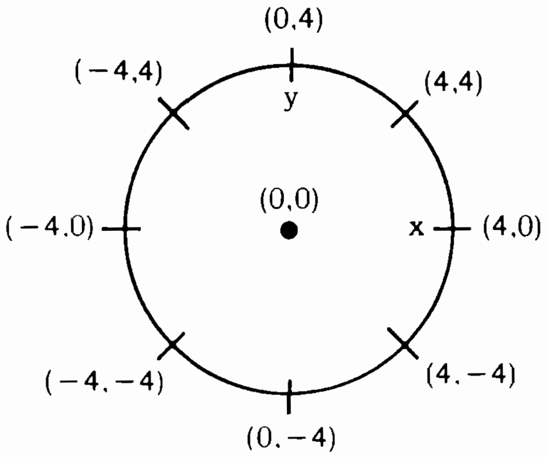
CALL JOYST *(key-unit,x-return,y-return)*

The JOYST subprogram allows you to input information to the computer based on the position of the lever on the Wired Remote Controllers accessory (available separately).

The *key-unit* is a numeric expression which, when evaluated, has a value of 1 through 4.

- 1 = controller 1
- 2 = controller 2
- 3 and 4 = reserved for possible future devices

Numeric variables must be used for *x-return* and *y-return*. The subprogram assigns an integer value of −4, +4, or 0 to each of these variables, based on the position of the joystick at that time, as shown below. The first value in parentheses is *x-return* and the second value is *y-return*.



You may then use these values in your program by referring to the variable names.

You will find more detailed instructions in the manual enclosed with the optional remote controls.

```
>NEW

>100 CALL CLEAR
>110 CALL CHAR(42,"FFFFFFFFFF
 FFFFFF")
>120 INPUT "SCREEN COLOR?":S
>130 INPUT "BLOCK COLOR?":F
>140 CALL CLEAR
>150 CALL SCREEN(S)
>160 CALL COLOR(2,F,1)
>170 CALL JOYST(2,X,Y)
>180 A=X*2.2+16.6
>190 B=Y*1.6+12.2
>200 CALL HCHAR(B,A,42)
>210 GOTO 170
>RUN

 --screen clears

 SCREEN COLOR?14
 BLOCK COLOR?9

 --screen clears

--color block will move
  around screen as joystick
  controller is moved

 (Press SHIFT C to stop the
      program)
```

KEY subprogram

CALL KEY *(key-unit, return-variable, status-variable)*

The KEY subprogram allows you to transfer one character from the keyboard directly to your program. This eliminates the need for an INPUT statement and saves time in getting data from a single key into memory. Because the character represented by the key pressed is not displayed on the screen, the information already on the screen is not disturbed by performing the CALL KEY statement. The *key-unit,* which indicates which keyboard is the input device, is a numeric expression which, when evaluated, has a value 0 through 5, as shown below:

- 0 = console keyboard
- 1 = left side of console keyboard or remote control 1
- 2 = right side of console keyboard or remote control 2
- 3, 4, 5 = available for possible future devices

1 and 2 are used when you want to use the console keyboard as two separate smaller duplicate keyboards, or you are using the remote control firebuttons.

The *return-variable* must be a numeric variable. The computer will place in *return-variable* the numeric character code represented by the key pressed. If the unit used is the console keyboard (unit 0), the character codes are the normal ASCII codes (see page 163) and will range from 0-95. If you are using the split keyboard (unit 1 and/or unit 2), the character codes will be 0 through 19. See page 165 for a list of these codes and their corresponding characters.

The *status-variable* is a numeric variable which serves as an indicator to let you know what happened at the keyboard. The computer will return one of the following codes to the *status-variable* after performing the CALL KEY routine:

- +1 = a new key was pressed since the last performance of the CALL KEY routine
- −1 = the same key was pressed during the performance of CALL KEY as was pressed during the previous performance
- 0 = no key was pressed

You can then check this status indicator in your program to determine what action to take next, as shown in line 110 of the program on the right. Line 110 is a test that gives you time to find and press a different key before the computer continues on to the next statement.

Examples:

```
>NEW

>100 CALL KEY(0,KEY,STATUS)
>110 IF STATUS=0 THEN 100
>120 NOTE=KEY-64
>130 ON NOTE GOTO 250,270,150
 ,170,190,210,230
>140 GOTO 100
>150 NOTE=262
>160 GOTO 280
>170 NOTE=294
>180 GOTO 280
>190 NOTE=330
>200 GOTO 280
>210 NOTE=349
>220 GOTO 280
>230 NOTE=293
>240 GOTO 280
>250 NOTE=440
>260 GOTO 280
>270 NOTE=494
>280 CALL SOUND(100,NOTE,2)
>290 GOTO 100
>RUN

  --plays a different note on
    the scale as you press
    the corresponding key (A-G)

 (Press SHIFT C to stop
  the program)
```

ABS — Absolute Value

ABS*(numeric-expression)*

The absolute value function gives you the absolute value of the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for evaluating numeric expressions (page 40) are used here. If the argument is positive, then the absolute value function gives you the argument itself. If the argument is negative, the absolute value function gives you the negative of the argument. Thus, for an argument, X:

- If X≥0, ABS(X) = X
- If X<0, ABS(X) = -X
  (e.g., ABS(-3) = -(-3) = 3)

Examples:

```
>NEW

>100 A=-27.36
>110 B=9.7
>120 PRINT ABS(A);ABS(B)
>130 PRINT ABS(3.8);ABS(-4.5)

>140 PRINT ABS(-3*2)
>150 PRINT ABS(A*(B-3.2))
>160 END
>RUN
  27.36  9.7
  3.8  4.5
  6
  177.84

 ** DONE **
```

ATN — Arctangent

ATN*(numeric-expression)*

The arctangent function gives you the arctangent of the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for evaluating numeric expressions (page 40) are used here. Thus, ATN(x) gives you the angle (in radians) whose tangent is x. If you want to get the equivalent angle in degrees, you need to multiply the answer you get by (180/(4 *ATN(1))) or 57.295779513079 which is 180/π. The value given to you for the arctangent function is always in the range −π/2<ATN(x)< π/2.

```
>NEW

>100 PRINT ATN(.44)
>110 PRINT ATN(1E127)
>120 PRINT ATN(1E-129);ATN(0)

>130 PRINT ATN(.3)*57.2957795
 13079
>140 PRINT ATN(.3)*(180/(4*AT
 N(1)))
>150 END
>RUN
  .4145068746
  1.570796327
  0  0
  16.69924423
  16.69924423

 ** DONE **
```

Built-In Numeric Functions

Introduction

Many special purpose functions are built in to TI BASIC. The functions described in this section perform some of the frequently used arithmetic operations. Obtaining the equivalent results for these functions requires a lot of programming in BASIC. Thus, they have been built in to TI BASIC and made easy for you to use. Built-in functions which are used with strings are discussed in the Built-In String Functions section, page 125. In addition to the built-in functions, you can also define your own functions (see User-Defined Functions, page 130).

INT — Integer

INT*(numeric-expression)*

The integer function gives you the largest integer that is not greater than the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for evaluating numeric expressions (page 40) are used here. The integer function always gives you the closest integer which is to the left of the number specified on the number line. Thus, for positive numbers, the decimal portion is dropped; for negative numbers, the next smallest integer value is used (i.e., INT(-2.3) = -3). If you specify an integer, then the same integer is given.

Examples:

```
>NEW

>100 B=.678
>110 A=INT(B*100+.5)/100
>120 PRINT A;INT(B)
>130 PRINT INT(-2.3);INT(2.2)

>140 STOP
>RUN
  .68  0
 -3  2

 ** DONE **
```

LOG — Natural Logarithm

LOG*(numeric-expression)*

The natural logarithm function gives you the natural logarithm of the number specified by the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here. The natural logarithm of x is usually shown as: $\log_e(x)$. The logarithm function is the inverse of the exponential function (EXP), see page 119. Thus, X =LOG(EXP(X)).

The argument of the natural logarithm function must be greater than zero. If you specify a value for the argument which is less than or equal to zero, the message "BAD ARGUMENT" is displayed, and the program stops running.

If you want to find the logarithm of a number in another base, B, use this formula.

$$\log_B(X) = \log_e(X)/\log_e(B)$$

For example, $\log_{10}(3) = \log_e(3)/\log_e(10)$

```
>NEW

>100 A=3.5
>110 PRINT LOG(A);LOG(A*2)
>120 PRINT LOG(EXP(2))
>130 STOP
>RUN
  1.252762968  1.945910149
  2.

 ** DONE **


>PRINT LOG(-3)

 * BAD ARGUMENT


>PRINT LOG(3)/LOG(10)
  .4771212547
```

COS — Cosine

COS*(numeric-expression)*

The cosine function gives you the cosine of the argument, x, where x is an angle in radians. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for evaluating numeric expressions (page 40) are used here. If the angle is in degrees, multiply the degrees by $\pi/180$ to get the equivalent angle in radians. You may use (4*ATN(1))/180 or 0.01745329251994 for $\pi/180$. Note that if you enter a value of x where $|x| \geq 1.5707963266375*10^{10}$, the message "BAD ARGUMENT" is displayed and the program stops running.

Examples:

```
>NEW

>100 A=1.047197551196
>110 B=60
>120 C=.01745329251994
>130 PRINT COS(A);COS(B*C)
>140 PRINT COS(B*(4*ATN(1))/1
 80)
>150 END
>RUN
  .5  .5
  .5

 ** DONE **

>PRINT COS(2.2E11)

 * BAD ARGUMENT
```

EXP — Exponential

EXP*(numeric-expression)*

The exponential function gives you the value of e^x, where e=2.718281828. The argument, x, is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here. The exponential function is the inverse of the natural logarithm function (LOG), see page 120. Thus, X=EXP(LOG(X)).

```
>NEW

>100 A=3.79
>110 PRINT EXP(A);EXP(9)
>120 PRINT EXP(A*2)
>130 PRINT EXP(LOG(2))
>140 END
>RUN
  44.25640028  8103.083928
  1958.628965
  2

 ** DONE **
```

RND — Random Number

RND

The random number function gives you the next pseudo-random number in the current sequence of pseudo-random numbers. The random number generated will be greater than or equal to zero and less than one. The sequence of random numbers generated by the random number function is the same every time the program is run unless the RANDOMIZE statement (page 121) appears in the program.

If you wish to obtain random integers between two values A and B (A<B), inclusive, use this formula:

INT((B−A+1)*RND)+A

Examples:

```
>NEW

>100 FOR I=1 TO 5
>110 PRINT INT(10*RND)+1
>120 NEXT I
>130 END
>RUN
  6
  4
  6
  4
  3

 ** DONE **
```

```
>NEW

>100 REM RANDOM INTEGERS
 BETWEEN 1 AND 20,INCLUSIVE
>110 FOR I=1 TO 5
>120 C=INT(20*RND)+1
>130 PRINT C
>140 NEXT I
>150 END
>RUN
  11
  8
  11
  8
  6

 ** DONE **
```

RANDOMIZE Statement

RANDOMIZE [*seed*]

The RANDOMIZE statement is used in conjunction with the random number function (RND). When the RANDOMIZE statement is not used, the random number function will generate the same sequence of pseudo-random numbers each time the program is run. When the RANDOMIZE statement is used without a *seed*, a different and unpredictable sequence of random numbers is generated by the random number function each time the program is run. If you use the RANDOMIZE statement with a *seed* specified, then the sequence of random numbers generated by the random number function depends upon the value of the *seed*. If the same *seed* is used each time the program is run, then the same sequence of numbers is generated. If a different *seed* is used each time the program is run, then a different sequence of numbers is generated. The *seed* may be any numeric expression. The number actually used for the *seed* is the first two bytes of the internal representation of the number. (See Appendix, page 173 for a complete explanation.) Thus, it is possible that the same sequence of numbers may be generated even if you specify different *seeds*. For example, RANDOMIZE 1000 and RANDOMIZE 1099 will produce the same first two bytes internally and thus will produce the same sequence of numbers. If the *seed* you specify is not an integer, then the value used is INT *(seed)* (see the Integer Function, page 120).

Examples:

```
>NEW

>100 RANDOMIZE 23
>110 FOR I=1 TO 5
>120 PRINT INT(10*RND)+1
>130 NEXT I
>140 STOP
>RUN
  6
  4
  3
  8
  8

 ** DONE **
```

SQR — Square Root Function

SQR*(numeric-expression)*

The square root function gives you the positive square root of the value specified by the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here. SQR(x) is equivalent to x ∧(1/2). The value specified by the argument may not be negative. If you specify a value for the argument which is less than zero, then the message "BAD ARGUMENT" is displayed and the program stops running.

Examples:

```
>NEW

>100 PRINT SQR(4);4^(1/2)
>110 PRINT SQR(10)
>120 END
>RUN
  2  2
  3.16227766

 ** DONE **

>PRINT SQR(-5)

 * BAD ARGUMENT
```

TAN — Tangent

TAN*(numeric-expression)*

The tangent function gives you the tangent of the argument, x, where x is an angle in radians. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for evaluating numeric expressions (page 40) are used here. If the angle is in degrees, multiply the degrees by $\pi/180$ to get the equivalent angle in radians. You may use (4 *ATN(1))/180 or 0.01745329251994 for $\pi/180$. Note that if you enter a value of x where $|x| \geq 1.5707963266375*10^{10}$, the message "BAD ARGUMENT" is displayed and the program stops running.

```
>NEW

>100 A=.7853981633973
>110 B=45
>120 C=.01745329251994
>130 PRINT TAN(A);TAN(B*C)
>140 PRINT TAN(B*(4*ATN(1))/1
80)
>150 END
>RUN
  1.  1.
  1

 ** DONE **

>PRINT TAN(1.76E10)

 * BAD ARGUMENT
```

SGN — Signum (Sign)

SGN*(numeric-expression)*

The signum function gives you the algebraic sign of the value specified by the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here. The signum function gives different values depending on the value of the argument. These values are given here. For argument, X:

- X<0, SGN(X) = −1
- X =0, SGN(X)=0
- X >0, SGN(X)=1

Examples:

```
>NEW

>100 A=-23.7
>110 B=6
>120 PRINT SGN(A);SGN(0);SGN(
 B)
>130 PRINT SGN(-3*3);SGN(B*2)

>140 END
>RUN
 -1  0  1
 -1  1

  ** DONE **
```

SIN — Sine

SIN*(numeric-expression)*

The sine function gives you the sine of the argument, x, where x is an angle in radians. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for evaluating numeric expressions (page 40) are used here. If the angle is in degrees, simply multiply the degrees by $\pi/180$ to get the equivalent angle in radians. You may use (4 *ATN(1))/180 or 0.01745329251944 for $\pi/180$. Note that if you enter a value of x where $| x | \geq 1.5707963266375 * 10^{10}$, the message "BAD ARGUMENT" is displayed and the program stops running.

```
>NEW

>100 A=.5235987755982
>110 B=30
>120 C=.01745329251994
>130 PRINT SIN(A);SIN(B*C)
>140 PRINT SIN(B*(4*ATN(1))/1
 80)
>150 END
>RUN
  .5  .5
  .5

  ** DONE **


>PRINT SIN(1.9E12)

  * BAD ARGUMENT
```

ASC — ASCII Value

ASC(*string-expression*)

The ASCII value function will give you the ASCII character code which corresponds to the first character of the string specified by the *string-expression* (see page 43). A list of the ASCII character codes for each character in the standard character set is given on page 163. The ASCII value function is the inverse of the character function (CHR$), see below.

Examples:

```
>NEW

>100 A$="HELLO"
>110 C$="JACK SPRAT"
>120 C=ASC(C$)
>130 B$="THE ASCII VALUE OF "

>140 PRINT B$;"H IS";ASC(A$)
>150 PRINT B$;"J IS";C
>160 PRINT B$;"N IS";ASC("NAM
 E")
>170 PRINT B$;"1 IS";ASC("1")

>180 PRINT CHR$(ASC(A$))
>190 END
>RUN
  THE ASCII VALUE OF H IS 72
  THE ASCII VALUE OF J IS 74
  THE ASCII VALUE OF N IS 78
  THE ASCII VALUE OF 1 IS 49
  H

 ** DONE **
```

CHR$ — Character

CHR$*(numeric-expression)*

The character function gives you the character corresponding to the ASCII character code specified in the argument. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here. The character function is the inverse of the ASCII value function (ASC), see above. If the argument specified is not an integer, then it is rounded to obtain an integer. A list of the ASCII character codes for each character in the standard character set is given on page 163. If the argument specified is a value between 32 and 95, inclusive, then a standard character is given. If the argument specified is between 96 and 159, inclusive, and a special graphics character has been defined for that value, then the graphics character is given. If you specify an argument which designates an undefined character (i.e., not a standard character or a defined graphics character), then the character given is whatever is in memory at that time.

If you specify a value for the argument which is less than zero or greater than 32767, the message "BAD VALUE" is displayed, and the program stops running.

```
>NEW

>100 A$=CHR$(72)&CHR$(73)&CHR
 $(33)
>110 PRINT A$
>120 CALL CHAR(97,"0103070F1F
 3F7FFF")
>130 PRINT CHR$(32);CHR$(97)
>140 PRINT CHR$(3*14)
>150 PRINT CHR$(ASC("+"))
>160 END
>RUN
 HI!

 *
 +

 ** DONE **

>PRINT CHR$(33010)

 * BAD VALUE
```

Built-In String Functions

Introduction

In addition to the built-in numeric functions, many other functions are built into TI BASIC. The functions discussed in this section are called string functions. String functions either use a string in some way to produce a numeric result, or the result of the evaluation of the function is a string. As you use your computer, you will find many ways to use the string functions described here. You can also define your own string functions (see User-Defined Functions, page 130). Note that any string function with a name that ends with a dollar sign (e.g. CHR$) always gives a string result and cannot be used in numeric expressions.

SEG$ — String Segment

SEG$*(string-expression,numeric-expression1,numeric-expression2)*

The string segment function gives you a portion (substring) of the string designated by the *string-expression*. *Numeric-expression1* identifies the position of the character in the original string which is the first character of the substring. The position of the first character in the string specified is position one. The length of the substring is specified by *numeric-expression2*. The normal rules for the evaluation of numeric expressions (page 40) and string expressions (page 43) are used here.

For this discussion, A$ is used for *string-expression*, X is used for *numeric-expression1* and Y is used for *numeric-expression2*. If you specify a value for X which is greater than the length of A$ (line 110) or a value of zero for Y (line 120), then you are given the null string. If you specify a value for Y which is greater than the remaining length in A$ starting at the position specified by X (line 130), then you are given the rest of A$ starting at the position specified by X.

If you specify a value for X which is less than or equal to zero, and/or specify a value for Y which is less than zero, then the message "BAD VALUE" is displayed and the program stops running.

Examples:

```
>NEW

>100 MSG$="HELLO THERE! HOW A
 RE YOU?"
>110 REM SUBSTRING BEGINS IN
 POSITION 14 AND HAS A LENGTH
  OF 12.
>120 PRINT SEG$(MSG$,14,12)
>130 END
>RUN
 HOW ARE YOU?

 ** DONE **
```

```
>NEW

>100 MSG$="I AM A COMPUTER."
>110 PRINT SEG$(MSG$,20,1)
>120 PRINT SEG$(MSG$,10,0)
>130 PRINT SEG$(MSG$,8,20)
>140 END
>RUN


 COMPUTER.

 ** DONE **
```

```
>PRINT SEG$(MSG$,-1,10)

 * BAD VALUE
```

LEN — Length

LEN(*string-expression*)

The length function gives you the number of characters in the string specified by the argument. The argument is the string value obtained when the *string-expression* is evaluated. The normal rules for the evaluation of string expressions (page 43) are used here. The length of a null string is zero. Remember that a space is a character and counts as part of the length.

Examples:

```
>NEW

>100 NAME$="CATHY"
>110 CITY$="NEW YORK"
>120 MSG$="HELLO "&"THERE!"
>130 PRINT NAME$;LEN(NAME$)
>140 PRINT CITY$;LEN(CITY$)
>150 PRINT MSG$;LEN(MSG$)
>160 PRINT LEN(NAME$&CITY$)
>170 PRINT LEN("HI!")
>180 STOP
>RUN
 CATHY  5
 NEW YORK  8
 HELLO THERE!  12
  13
  3

 ** DONE **
```

POS — Position

POS(*string-1,string-2,numeric-expression*)

The position function finds the first occurrence of *string-2* within *string-1*. Both *string-1* and *string-2* are string expressions. The *numeric-expression* is evaluated and rounded, if necessary, to obtain an integer, *n*. The normal rules for the evaluation of string expressions (page 43) and numeric expressions (page 40) are used here. The search for *string-2* begins at the *n*th character of *string-1*. If *string-2* is found, then the character position within *string-1* of the first character of *string-2* is given. If *string-2* is not found, then a value of zero is given. The position of the first character in *string-1* is position one. If you specify a value for *n* which is greater than the number of characters in *string-1*, then a value of zero is given. If the value specified for *n* is less than zero, the message "BAD VALUE" is displayed and the program stops running.

```
>NEW

>100 MSG$="HELLO THERE! HOW A
 RE YOU?"
>110 PRINT "H";POS(MSG$,"H",1
 )
>120 C$="RE"
>130 PRINT C$;POS(MSG$,C$,1);
 POS(MSG$,C$,12)
>140 PRINT "HI";POS(MSG$,"HI"
 ,1)
>150 END
>RUN
 H  1
 RE  10  19
 HI  0

 ** DONE **
```

User-Defined Functions

Introduction

In addition to the built-in functions described in the two previous sections, TI BASIC provides user-defined functions. User-defined functions can simplify programming by avoiding repeated use of complicated expressions. Once a function has been defined using the DEF statement, it may be used anywhere in the program by referencing the name you gave to the function.

STR$ — String-Number

STR$(*numeric-expression*)

The string-number function converts the number specified by the argument into a string. The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here. When the number is converted into a string, the string is a valid representation of a numeric constant with no leading or trailing spaces. For example, if B=69.5, then STR$ (B) is the string "69.5." Only string operations may be performed on the strings created using the string-number function. The string-number function is the inverse of the value function (VAL); see below. In the example, note that leading and trailing spaces are not present on the numbers converted to strings.

Examples:

```
>NEW

>100 A=-26.3
>110 PRINT STR$(A);" ";A
>120 PRINT 15.7;STR$(15.7)
>130 PRINT STR$(VAL("34.8"))
>140 END
>RUN
 -26.3 -26.3
  15.7 15.7
 34.8

 ** DONE **
```

VAL — Value

VAL(*string-expression*)

The value function is the inverse of the string-number function (STR$); see above. If the string specified by the *string-expression* is a valid representation of a numeric constant, then the value function converts the string to a numeric constant. For example, if A$ ="1234", then VAL(A$)=1234. The normal rules for the evaluation of string expressions (page 43) are used here. If the string specified is not a valid representation of a number or if the string is of zero length, then the message "BAD ARGUMENT" is displayed and the program stops running. If you specify a string which is longer than 254 characters, the message "BAD ARGUMENT" is displayed and the program stops running.

```
>NEW

>100 P$="23.6"
>110 N$="-4.7"
>120 PRINT VAL(P$);VAL(N$)
>130 PRINT VAL("52"&".5")
>140 PRINT VAL(N$&"E"&"12")
>150 PRINT STR$(VAL(P$))
>160 END
>RUN
  23.6 -4.7
  52.5
 -4.7E+12
 23.6

 ** DONE **
```

DEF

The *parameter* used in the DEF statement is local to the DEF statement in which it is used. This means that it is distinct from any variable with the same name which is used in other statements in the program. Thus, evaluating the function does not affect the value of a variable which has the same name as the *parameter*.

Examples:

```
>NEW

>100 DEF FUNC(A)=A*(A+B-5)
>110 A=6.9
>120 B=13
>130 PRINT "B= ";B:"FUNC(3)=
 ";FUNC(3):"A= ";A
>140 END
>RUN
 B=  13
 FUNC(3)=  33
 A=  6.9

 ** DONE **
```

A DEF statement is only performed when the function it defines is referenced in an expression. When the computer encounters a DEF statement while running a program, it takes no action but proceeds to the next statement. A DEF statement may appear anywhere in a program and need not logically precede a reference to the function, but the function definition must have a lower line number than any statement which references the function. A DEF statement can reference other defined functions (line 170).

```
>NEW

>100 REM FIND F'(X) USING
 NUMERICAL APPROXIMATION
>110 INPUT "X=? ":X
>120 IF ABS(X)>.01 THEN 150
>130 H=.00001
>140 GOTO 180
>150 H=.001*ABS(X)
>160 DEF F(Z)=3*Z^3-2*Z+1
>170 DEF DER(X)=(F(X+H)-F(X-H
 ))/(2*H)
>180 PRINT "F'(";STR$(X);")=
 ";DER(X)
>190 END
>RUN
 X=? .1
 F'(.1)= -1.90999997

 ** DONE **
```

In a DEF statement, the function you specify may not reference itself either directly (e.g. DEF B=B*2) or indirectly (e.g. DEF F=G; DEF G = F).The *parameter* you specify may not be used as an array. You can use an array element in a function definition as long as the array does not have the same name as the *parameter*.

```
>NEW

>100 DEF GX(X)=GX(2)*X
>110 PRINT GX(3)
>120 END
>RUN

 * MEMORY FULL IN 110

>100 DEF GX(A)=A(3)^2
>RUN

 * NAME CONFLICT IN 100
```

DEFine

DEF { *numeric-function-name* [*(parameter)*] = *numeric-expression*
 string-function-name [*(parameter)*] = *string-expression* }

The DEFine statement allows you to define your own functions to use within a program. The *function-name* you specify may be any valid variable name (see page 39). If you specify a *parameter* following the *function-name*, the *parameter* must be enclosed in parentheses and may be any valid variable name. Note that if the expression you specify evaluates to a string result, the *function-name* you use must be a string variable name (i.e., the last character must be a dollar sign, $).

The DEFine statement specifies the function to be used based upon the *parameter* (if specified), variables, constants, and other built-in functions. Once a function has been defined, you may use the function in any string or numeric expression by entering the *function-name*. The *function-name* must be followed by an argument enclosed in parentheses if a *parameter* was specified in the DEF statement. If a function has no *parameter* specified, when a reference to the function is encountered in an expression, the function is evaluated using the current values of the variables which appear in the DEF statement.

If you specify a *parameter* for a function, when a reference to the function is encountered in an expression, the argument is evaluated and its value is assigned to the *parameter*. The expression in the DEF statement is then evaluated using the newly assigned value of the *parameter* and the current values of the other variables in the DEF statement.

Examples:

```
>NEW

>100 DEF PI=4*ATN(1)
>110 PRINT COS(60*PI/180)
>120 END
>RUN
  .5

 ** DONE **
```

```
>NEW

>100 REM EVALUATE Y=X*(X-3)
>110 DEF Y=X*(X-3)
>120 PRINT " X  Y"
>130 FOR X=-2 TO 5
>140 PRINT X;Y
>150 NEXT X
>160 END
>RUN
  X  Y
 -2  10
 -1  4
  0  0
  1 -2
  2 -2
  3  0
  4  4
  5  10

 ** DONE **
```

```
>NEW

>100 REM TAKE A NAME AND
 PRINT IT BACKWARDS
>110 DEF BACK$(X)=SEG$(NAME$,
 X,1)
>120 INPUT "NAME? ":NAME$
>130 FOR I=LEN(NAME$) TO 1 ST
 EP -1
>140 BNAME$=BNAME$&BACK$(I)
>150 NEXT I
>160 PRINT NAME$:BNAME$
>170 END
>RUN
 NAME? ROBOT
 ROBOT
 TOBOR

 ** DONE **
```

Arrays

Introduction

An array is a collection of variables arranged in a way that allows you to use them easily in a computer program. The most common way of grouping variables is in a list, which is called a one-dimensional array. Each variable in the list is called an element of the array. The length of the list is limited only by the amount of memory available.

By using the array capability of TI BASIC you can do many things with a list – you can print the elements forward or backward, rearrange them, add them together, multiply them, or select certain ones for processing.

In TI BASIC an array may begin with element 0 or element 1. By using the OPTION BASE statement, you control which beginning element the computer establishes (see page 138). For consistency in describing arrays, we are assuming that the first element in each array is element 1.

Let's say you want to use the computer to take two lists of four numbers and print all possible combinations of the numbers in both lists. You might call the first array X and the second one Y. Since X and Y name a *collection* of numbers, rather than a single variable, the computer needs a way to refer to the individual elements in each array. You must supply a pointer, called a subscript, to the particular element in the array you want the computer to use. This subscript is enclosed in parentheses and always immediately follows the name of the array. The subscript may be explicit, such as X(3), which refers to the third element in list X, or it may be a variable, as in X(T), where the value of T points to the proper element. In any case, the subscript is always either a positive integer or zero.

The program on the right pairs the numbers in array X and array Y. Notice that by using the array technique only a few program lines are needed for this relatively complex procedure.

Multi-Dimensional Arrays

With TI BASIC you can extend your use of arrays to include tabular information, arranged in rows and columns, called two-dimensional arrays. You can think of the TIC-TAC-TOE game as an example of a two-dimensional array.

X	O	X
O	X	X
X	O	O

Examples:

```
>NEW

>100 REM THIS PROGRAM PAIRS
 TWO LISTS
>110 REM LINES 120 TO 150
 ASSIGN VALUES TO LIST X
>120 FOR T=1 TO 4
>130 READ X(T)
>140 NEXT T
>150 DATA 1,3,5,7
>160 REM LINES 170 TO 200
 ASSIGN VALUES TO LIST Y
>170 FOR S=1 TO 4
>180 READ Y(S)
>190 NEXT S
>200 DATA 2,4,6,8
>210 REM LINES 220 TO 270
 PAIR THE LISTS AND PRINT
 THE COMBINATIONS
>220 FOR T=1 TO 4
>230 FOR S=1 TO 4
>240 PRINT X(T);Y(S);"  ";
>250 NEXT S
>260 PRINT
>270 NEXT T
>280 END
>RUN
  1  2   1  4   1  6   1  8
  3  2   3  4   3  6   3  8
  5  2   5  4   5  6   5  8
  7  2   7  4   7  6   7  8

 ** DONE **
```

DEF

If you specify a *parameter* when defining a function, you must specify an argument when you reference the function. Similarly, if you do not specify a *parameter* when defining a function, you cannot specify an argument in the function reference.

Examples:

```
>NEW

>100 DEF SQUARE(X)=X*X
>110 PRINT SQUARE
>120 END
>RUN

 * NAME CONFLICT IN 110

>100 DEF PI=3.1416
>110 PRINT PI(2)
>RUN

 * NAME CONFLICT IN 110
```

DIMension

DIM {*array-name (integer1[,integer2][,integer3])*},. . .

The DIMension statement reserves space for both numeric and string arrays. You can explicitly dimension an array only once in your program. If you dimension an array, the DIM statement must appear in the program before any other reference to the array. If you dimension more than one array in a single DIM statement, the array names must be separated by commas. The *array-name* may be any valid variable name.

You may use one, two, or three-dimensional arrays in TI BASIC (see page 134 for an explanation of arrays and their uses). The number of values in parentheses following the array name tells the computer how many dimensions the array has.

One-dimensional arrays have only one integer value following their name. Two-dimensional arrays are described with two integer values which define the number of rows and columns. Three-dimensional arrays have three integer values defining their characteristics.

- DIM A(6) – describes a one-dimensional array.
- DIM A(12,3) – describes a two-dimensional array.
- DIM A(5,2,11) – describes a three-dimensional array.

If an array is not dimensioned in a DIM statement, the computer will automatically assign a value of 10 for *integer1* (and a value of 10 for *integer2* and *integer3* if needed) for each array used.

Space is allocated for your array after you enter the RUN command but before the program is actually run. Each element in a string array, however, is a null string until you actually place values in each element. If your computer memory cannot handle an array with the dimensions you specified, you will get a "MEMORY FULL" message and your program will not run.

Examples:

```
>DIM A(12),B(5)
```

```
>NEW

>100 DIM X(15)
>110 FOR I=1 TO 15
>120 READ X(I)
>130 NEXT I
>140 REM PRINT LOOP
>150 FOR I=15 TO 1 STEP -1
>160 PRINT X(I);
>170 NEXT I
>180 DATA 1,2,3,4,5,6,7,8,9,1
 0,11,12,13,14,15
>190 END
>RUN
  15  14  13  12  11  10  9
  8  7  6  5  4  3  2  1
 ** DONE **
```

You can represent the gameboard with this array:

T(1,1)	T(1,2)	T(1,3)
T(2,1)	T(2,2)	T(2,3)
T(3,1)	T(3,2)	T(3,3)

As in the one-dimensional arrays described earlier, you refer to a two-dimensional element with a subscript, in this case a double-subscript to refer to the row and column location. Often you will use a variable as a subscript, rather than an explicit subscript; for example T(R,C).

When you use a two-dimensional array, you will often use nested FOR-NEXT loops. One loop will take the computer through the rows and the other will take it through the columns. The program on the right creates a two-dimensional array — a multiplication table — with five rows and five columns, using nested FOR-NEXT loops.

You can work with arrays of one, two, or three dimensions on your TI Home Computer. Elements in three-dimensional arrays are referenced with three subscript values: X(22,14,7) or M(I,J,K).

Examples:

```
>NEW

>100 REM MULTIPLICATION TABLE

>110 CALL CLEAR
>120 CALL CHAR(96,"FF")
>130 CALL CHAR(97,"8080808080
 808080")
>140 CALL CHAR(98,"FF80808080
 808080")
>150 FOR A=1 TO 5
>160 FOR B=1 TO 5
>170 M(A,B)=A*B
>180 NEXT B
>190 NEXT A
>200 FOR A=1 TO 5
>210 FOR B=1 TO 5
>220 PRINT M(A,B);
>230 IF B<>1 THEN 250
>240 PRINT CHR$(97);" ";
>250 NEXT B
>260 PRINT
>270 REM THE FOLLOWING
 STATEMENTS PRINT THE LINES
 DEFINING THE TABLE
>280 IF A<>1 THEN 330
>290 PRINT
>300 CALL HCHAR(23,3,96,3)
>310 CALL HCHAR(23,6,98)
>320 CALL HCHAR(23,7,96,16)
>330 NEXT A
>340 END
>RUN
```

```
 -- screen clears

 1 | 2  3  4  5
___|_____________
   |
 2 | 4  6  8  10
 3 | 6  9  12  15
 4 | 8  12  16  20
 5 | 10  15  20  25

 ** DONE **
```

OPTION BASE

OPTION BASE $\left\{\begin{matrix}0\\1\end{matrix}\right\}$

The OPTION BASE statement allows you to set the lower limit of array subscripts at one instead of zero. You can omit the OPTION BASE statement if you want the lower limit of the subscripts to be zero.

If you include an OPTION BASE statement in your program, you must give it a lower line number than any DIMension statement (see page 136) or any reference to an element in any array. You may have only *one* OPTION BASE statement in a program, and it applies to *all* array subscripts in your program. Therefore, you cannot have one array subscript beginning with 0 and another beginning with 1 in the same program.

If you use some integer other than one or zero in the OPTION BASE statement, the computer will stop the program and print "INCORRECT STATEMENT."

Examples:

```
>NEW

>100 OPTION BASE 1
>110 DIM X(5,5,5)
>120 X(1,0,1)=3
>130 PRINT X(1,0,1)
>140 END
>RUN

 * BAD SUBSCRIPT IN 120

>100 ENTER
>RUN
  3

 ** DONE **
```

DIM

Subscripting An Array

Anytime you want to reference an array in your program, you must be specific about which element in the array you want the computer to use. To do this, you point to the element with a *subscript*. Subscripts are enclosed in parentheses immediately following the name of the array. A subscript can be any valid numeric expression which evaluates to a non-negative result. This result will be rounded to the nearest integer, if necessary.

The number of elements reserved for an array determines the maximum value of each subscript for that array. If you are using an array not defined in a DIMension statement, the maximum value of each subscript is 10. The minimum value is zero, unless an OPTION BASE statement (see page 138) sets the minimum subscript value at 1. Thus, an array defined as DIM A(6) actually has seven accessible elements in TI BASIC, unless the zero subscript is eliminated by the OPTION BASE 1 statement.

The example on the right assumes that the array begins with element 1 (OPTION BASE 1 on line 120):

- line 130 – This line defines T as a one-dimensional array with 25 elements.
- line 160 – The numeric variable I here subscripts T. Whatever value I contains at this time will be used to point to an element of T. If I=3, the third element of T will be added.
- line 200 – The subscript 14 tells the computer to print the fourteenth element of T.
- line 220 – The computer will evaluate the numeric expression N+2. If N=15 at this time, the seventeenth element of T will be printed.

If you access an array with a subscript greater than the maximum number of elements defined for that array, or if your subscript has a zero value and you used an OPTION BASE 1 statement, a "BAD SUBSCRIPT" message will print and the program will end.

(See page 135 for subscripting multi-dimensional arrays.)

Examples:

```
>NEW

>100 REM DEMO OF DIM AND
 SUBSCRIPTS
>110 S=100
>120 OPTION BASE 1
>130 DIM T(25)
>140 FOR I=1 TO 25
>150 READ T(I)
>160 A=S+T(I)
>170 PRINT A;
>180 NEXT I
>190 PRINT::
>200 PRINT T(14)
>210 INPUT "ENTER A NUMBER BE
 TWEEN 1 AND 23:":N
>220 PRINT T(N+2)
>230 DATA 12,13,43,45,65,76,7
 8,98,56,34,23,21,100,333,222
 ,111,444,666,543,234,89,765,
 90,101,345
>240 END
>RUN
  112  113  143  145  165
  176  178  198  156  134
  123  121  200  443  322
  211  544  766  643  334
  189  865  190  201  445

  333
 ENTER A NUMBER BETWEEN 1 AND
  23:14
  111

 ** DONE **
```

GOSUB

$\left\{\begin{array}{l}\text{GOSUB}\\\text{GO SUB}\end{array}\right\}$ *line-number*

The GOSUB statement is used with the RETURN statement (see page 142) to allow you to transfer the program to a subroutine, complete the steps in the subroutine, and return to the next program line following the GOSUB statement. When the computer performs the GOSUB statement, it saves the next line number of the main program so that it can return to that point when it encounters a RETURN statement in the subroutine.

(The space between GO and SUB is optional.)

Examples:

```
>NEW

>100 REM BUILD AN ARRAY,
 MULTIPLY EACH ELEMENT BY 3,
 PRINT BOTH ARRAYS
>110 FOR X=1 TO 4
>120 FOR Y=1 TO 7
>130 I(X,Y)=INT(30*RND)+1
>140 NEXT Y
>150 NEXT X
>160 PRINT "FIRST ARRAY":
>170 GOSUB 260
>180 FOR X=1 TO 4
>190 FOR Y=1 TO 7
>200 I(X,Y)=3*I(X,Y)
>210 NEXT Y
>220 NEXT X
>230 PRINT "3 TIMES VALUES IN
 FIRST ARRAY"::
>240 GOSUB 260
>250 STOP
>260 REM SUBROUTINE TO PRINT
 ARRAY
>270 FOR X=1 TO 4
>280 FOR Y=1 TO 7
>290 PRINT I(X,Y);
>300 NEXT Y
>310 PRINT
>320 NEXT X
>330 PRINT
>340 RETURN
>RUN
 FIRST ARRAY

  16  12  17  12  8  17  8
  18  22  1  29  16  14  11
  5  25  22  4  24  11  24
  26  21  18  2  12  20  15

 3 TIMES VALUES IN FIRST ARRA
 Y

  48  36  51  36  24  51  24
  54  66  3  87  48  42  33
  15  75  66  12  72  33  72
  78  63  54  6  36  60  45

 ** DONE **
```

Subroutines

Introduction

Subroutines may be thought of as separate self-contained programs within a main program. They usually perform a certain action, such as printing some information, performing a calculation, or reading values into an array. Putting these actions into a subroutine allows you to type that set of statements only once and then perform that set of statements from anywhere in the program with a GOSUB statement (see page 140).

The GOSUB statement initially behaves like a GOTO statement. It causes the computer to jump to the *line-number* listed. However, subroutine programming gives the computer the capability to "remember" where the branch occurred in the main program and return to that point when it finishes the subroutine. This technique requires that the last statement in the subroutine be a RETURN statement (see page 142). The program will normally have either a STOP statement or some other unconditional branching statement immediately before the subroutines so that the computer won't accidentally "fall into" the subroutines. The subroutines should be entered only by a GOSUB instruction and may be entered at any *line-number* within the subroutine.

The example on the right illustrates how the GOSUB and RETURN statements might be arranged in your program. The program begins running at line 100. At line 300 it skips to the first subroutine, performs lines 700 through 780, and returns to line 310. When it reaches line 400, it goes to the second subroutine, performs lines 900 through 980, returns to line 410, and continues running. At line 450 it again goes to subroutine 1, this time entering at line 750 and continuing to the RETURN. Then it goes back to the main program at line 460 and continues running. At line 480 it again jumps to the first subroutine, runs lines 700 through 780, returns to line 490, then stops running at line 600. The STOP statement in line 600 keeps the computer from performing the subroutines unless you specifically direct it there with a GOSUB.

Examples:

```
>NEW

>100 REM MAIN PROGRAM
     .
     .
     .
>300 GOSUB 700
>310 .
     .
     .
>400 GOSUB 900
>410 .
     .
     .
>450 GOSUB 750
>460 .
     .
     .
>480 GOSUB 700
>490 .
     .
     .
>600 STOP
>700 REM SUBROUTINE1
     .
     .
>750 .
     .
     .
>780 RETURN
>900 REM SUBROUTINE2
     .
     .
     .
>980 RETURN
>990 END
```

RETURN

RETURN

The RETURN statement is used with the GOSUB statement (see page 140) to provide a branch and return structure for TI BASIC. Whenever the computer encounters a RETURN statement, it takes the program back to the program line immediately following the GOSUB statement that transferred the computer to that particular subroutine in the first place. You can easily develop programs with subroutines which jump to other subroutines and back again, if you are careful that each GOSUB leads the computer to a RETURN statement (see page141 for example).

If, when running a program, the computer encounters a RETURN statement before performing a GOSUB instruction, the program will terminate with the message "CAN'T DO THAT."

Examples:

```
>NEW

>100 FOR I=1 TO 3
>110 GOSUB 150
>120 PRINT "I=";I
>130 NEXT I
>140 STOP
>150 REM SUBROUTINE
>160 FOR X=1 TO 2
>170 PRINT "X=";X
>180 NEXT X
>190 RETURN
>RUN
 X= 1
 X= 2
 I= 1
 X= 1
 X= 2
 I= 2
 X= 1
 X= 2
 I= 3

 ** DONE **
```

GOSUB

Within a subroutine, you may want the computer to jump to another subroutine, complete it, come back to the first subroutine, complete its steps, then return to the main program at the point where the original branch occurred. You can do this easily with the proper pairing of GOSUB and RETURN statements. However, be sure you exercise care in designing subroutines so that the computer will not "lose its place."

In the example on the right, the main program jumps to subroutine 1 when it reaches line 500. In subroutine 1, when the program reaches line 730, it goes to subroutine 2. When the RETURN in subroutine 2 is encountered (line 850), the computer returns to subroutine 1 at line 740, finishes the subroutine, returns to the main program and completes it through line 600.

If the GOSUB statement transfers the program to a *line-number* not in the program, the program will end and the message "BAD LINE NUMBER" will print. If the GOSUB transfers the program to its own *line-number*, the program will stop and the message "MEMORY FULL" will print.

Examples:

```
>NEW

>100 REM NESTED SUBROUTINES
>110 REM MAIN PROGRAM
     .
     .
     .
>500 GOSUB 700
>510 .
     .
     .
>600 STOP
>700 REM SUBROUTINE1
     .
     .
     .
>730 GOSUB 800
>740 .
     .
     .
>790 RETURN
>800 REM SUBROUTINE2
     .
     .
     .
>850 RETURN
```

```
>NEW

>100 X=12
>110 Y=23
>120 GOSUB 120
>130 PRINT Z
>140 STOP
>150 REM SUBROUTINE
>160 Z=X+Y*120/5
>170 RETURN
>RUN

* MEMORY FULL IN 120


>120 GOSUB 150
>RUN
  564

 ** DONE **
```

File Processing

Introduction

Your TI Home Computer has the capability to store both programs and data on accessory devices. You can later load and use these files with your computer as often as you wish, and delete them when you no longer need them.

The file processing capability of your computer offers you a powerful programming tool. You can eliminate retyping your favorite programs, save important information, and create procedures to update data important to you. TI BASIC provides an extensive range of file-processing features, including sequential and random file organization and processing, fixed and variable length records, and display and internal formats for data. This section describes the TI BASIC statements which use these features – OPEN, CLOSE, INPUT, PRINT, and RESTORE. As new accessory devices become available, the file features they use will be described in the accompanying manuals.

Audio Cassette Tape Recorders

Your TI Home Computer can process files from either one or two standard audio cassette tape recorders (see pages 15-16 of this book for instructions on attaching the recorders). These recorders are designated as CS1 and CS2. To save and/or load programs you need only one recorder. To read data from a file, process it in your program, and at the same time create a new data file, you will need two recorders – one to read the stored data and one to write the processed data.

Specific requirements for using file processing features with cassette recorders are given at the end of each statement description.

ON-GOSUB

ON *numeric-expression* {GOSUB | GO SUB} *line-number*[,*line-number*] . . .

The ON-GOSUB statement is used with the RETURN statement (see page 142) to tell the computer to perform one of several subroutines, depending on the value of a *numeric-expression*, and then go back to the main program sequence.

The computer first evaluates the *numeric-expression* and converts the result to an integer, rounding if necessary. This integer tells the program which subroutine *line-number* in the ON-GOSUB statement to perform next. If the value of the *numeric-expression* is 1, the computer will proceed to the first *line-number* listed in the ON-GOSUB statement. If the value is 2, the computer will branch to the second *line-number* given, and so on.

Additionally the computer will save the next line number following the ON-GOSUB statement and return to this point after performing the subroutine. The subroutine must contain a RETURN statement to signal the computer to go back to the saved line number and continue the program from that statement. Otherwise, the program will continue until it reaches the end, as if a GOTO was performed instead of a GOSUB.

If the rounded value of the *numeric-expression* is less than 1 or greater than the number of line numbers in the ON-GOSUB statement, the program will terminate with the message "BAD VALUE IN xx."

If the *line-number* listed is not a valid program line, the message "BAD LINE NUMBER" will print when you perform the statement.

Examples:

```
>NEW

>100 INPUT "CODE=?":CODE
>110 IF CODE=9 THEN 290
>120 INPUT "HOURS=?":HOURS
>130 ON CODE GOSUB 170,200,23
 0,260
>140 PAY=RATE*HOURS+BASEPAY
>150 PRINT "PAY IS $";PAY
>160 GOTO 100
>170 RATE=3.10
>180 BASEPAY=5
>190 RETURN
>200 RATE=4.25
>210 BASEPAY=25
>220 RETURN
>230 RATE=10
>240 BASEPAY=50
>250 RETURN
>260 RATE=25
>270 BASEPAY=100
>280 RETURN
>290 END
>RUN
 CODE=?4
 HOURS=?40
 PAY IS $ 1100
 CODE=?2
 HOURS=?37
 PAY IS $ 182.25
 CODE=?3
 HOURS=?35.75
 PAY IS $ 407.5
 CODE=?1
 HOURS=?40
 PAY IS $ 129
 CODE=?9

 ** DONE **
```

```
>RUN
 CODE=?5
 HOURS=?40

 * BAD VALUE IN 130
```

```
>130 ON CODE GOSUB 170,200,23
 0,600
>RUN
 CODE=?4
 HOURS=?40

 * BAD LINE NUMBER IN 130
```

OPEN

As additional accessory devices become available, you will find their *file-names* included in the manuals which accompany them.

- *file-organization* – Files used in TI BASIC can be organized either sequentially or randomly. Records on a sequential file are read or written one after the other in sequence from beginning to end. Random-access files (called RELATIVE in TI BASIC) can be read or written in any record order. They may also be processed sequentially.

  To indicate which logical structure a file has, enter either SEQUENTIAL or RELATIVE in the OPEN statement. You may optionally specify the initial number of records on a file by following the word SEQUENTIAL or RELATIVE with a numeric expression.

  If you omit the *file-organization* specification, the computer will assume SEQUENTIAL organization.

- *file-type* – This specification designates the format of the data stored on the file: DISPLAY or INTERNAL.

  The DISPLAY-type format refers to printable (ASCII) characters. The DISPLAY format is normally used when the output will be read by people, rather than by the computer. Each DISPLAY-type record usually corresponds to one print line.

  INTERNAL-type data is recorded in internal machine format which has not been translated into printable characters. Data in this form can be read easily by the computer but not by people. (See page 152 for a full explanation of how data is stored internally.)

  You will find that the INTERNAL format is more efficient for recording data on a storage device such as a cassette tape. It requires less space and is easier to format with a PRINT statement (see page 157 for directions on formatting PRINT statements for INTERNAL-type records and page 159 for DISPLAY-type records). Because the computer uses INTERNAL-type data internally, a program runs in less time when your data files are in INTERNAL format. The computer won't have to convert DISPLAY characters into INTERNAL format and back again.

  If this specification is omitted, the computer assumes DISPLAY format.

Examples:

```
>100 OPEN #4:"CS2",OUTPUT,INT
 ERNAL,SEQUENTIAL,FIXED
```

```
>120 OPEN #12:NAME$,RELATIVE
 50,INPUT,FIXED,INTERNAL
```

```
>100 OPEN #10:"CS1",OUTPUT,FI
 XED
```

```
 (computer assumes SEQUENTIAL,
  DISPLAY,PERMANENT)
```

OPEN

OPEN *#file-number:file-name[,file-organization][,file-type][,open-mode][,record-type][,file-life]*

The OPEN statement prepares a BASIC program to use data files stored on accessory devices. The OPEN statement does this by providing the necessary link between a *file-number* used in your program and the particular accessory device on which the file is located.

The OPEN statement describes a file's characteristics to the computer so that your program can process it or create it. With some accessory devices the computer will check that the file or device characteristics match the information specified in the OPEN statement for that file. If they don't match or the computer cannot find or create the file, the file will not be opened and an I/O error message will be printed (see error message table, page 171).

The *file-number* and *file-name* must be included in the OPEN statement. The other information can be included in any order or can be omitted. If you leave out any specification, the computer will assume certain standard characteristics for the file, called "defaults," as described later in this section.

- *file-number* – All TI BASIC statements which refer to files do so by means of a *file-number* between 0 and 255 inclusive. The *file-number* is assigned to a particular file by the OPEN statement. Since *file-number* 0 refers to the keyboard and screen of your computer and is always accessible, you cannot open or close *file-number* 0 in your program statements. You may assign the other numbers as you wish, as long as each open file in your program has a different number.

  The *file-number* is entered as the number sign (#) followed by a numeric expression. When the computer evaluates this expression and rounds the answer to the nearest integer, the number must be 1 to 255 inclusive and cannot be the same *file-number* as any other file you are using concurrently in the program.

- *file-name* – A *file-name* refers to a device or to a file located on a device, depending on the capability of the accessory. Each accessory has a predefined name which the computer recognizes. For example, the valid *file-names* for the two audio cassette recorders are "CS1" and "CS2." By including this *file-name* in the OPEN statement, you are telling the computer to access a particular file or device whenever the program references the associated *file-number*. The *file-name* can be any string expression which evaluates to a valid *file-name*. If you use a string constant, you must enclose it in quotes.

Examples:

```
>100 OPEN #2:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED 128,PE
 RMANENT
```

```
>100 OPEN #25:"CS1",SEQUENTIA
 L,INTERNAL,INPUT,FIXED,PERMA
 NENT
>110 X=100
>120 OPEN #X+5:"CS2",SEQUENTI
 AL,INTERNAL,OUTPUT,FIXED,PER
 MANENT
```

```
>130 N=2
>140 OPEN #122:"CS"&STR$(N),S
 EQUENTIAL,INTERNAL,OUTPUT,FI
 XED,PERMANENT
```

OPEN

Cassette Recorder Information

- *file-number** – any number between 1 and 255 inclusive
- *file-name** – "CS1" or "CS2"
- *file-organization* – SEQUENTIAL
- *file-type* – INTERNAL (preferred) or DISPLAY
- *open-mode** – INPUT or OUTPUT
- *record-type** – FIXED

*This specification is required.

For cassette tape records, you may specify any length up to 192 positions. However, the cassette tape device uses records with 64, 128, or 192 positions and will pad the record you specify to the appropriate length. Thus, if you specify an 83-position cassette record, the computer will actually write a 128-position record. If the record length is not specified, a 64-position record length is assumed.

For cassette devices, the computer does not compare the file specifications in the OPEN statement to the characteristics of an existing file.

Whenever the computer performs the OPEN statement for a cassette tape device, you will receive instructions on your screen for activating the recorder, as shown on the right.

Note: Only "CS1" can be specified for an INPUT file. Both "CS1" and "CS2" can be used for OUTPUT files.

Examples:

```
>NEW

>100 OPEN #2:"CS1",INTERNAL,I
 NPUT,FIXED
 .
 . program lines
 .
>290 CLOSE #2
>300 END
>RUN

 * REWIND CASSETTE TAPE    CS1
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY     CS1
   THEN PRESS ENTER
    .
    . rest of program run
    .

 * PRESS CASSETTE STOP     CS1
   THEN PRESS ENTER

 ** DONE **
```

OPEN

- *open-mode* – This entry instructs the computer to process the file in the INPUT, OUTPUT, UPDATE or APPEND mode. If you omit this clause, the computer will assume the UPDATE mode.
  - – INPUT files may be read only.
  - – OUTPUT files may be written only. The new file created will have all the characteristics given by the OPEN statement specifications and any standard defaults.
  - – UPDATE files may be both read and written. The usual processing is to read a record, change it in some way, then write the altered record back out on the file.
  - – APPEND mode allows data to be added at the end of the existing file. The records already on the file cannot be accessed in this mode.
- *record-type* – This entry specifies whether the records on the file are all the same length (FIXED) or vary in length (VARIABLE). The keyword FIXED or VARIABLE may be followed by a numeric expression specifying the maximum length of a record. Each accessory device has its own maximum record length, so be sure to check the manuals which accompany them. If you omit the record-length specification, the computer will assume a record length depending upon the device used.

  If you define a file as RELATIVE, you must use FIXED-length records. If this entry is omitted for RELATIVE files, FIXED-length records are assumed, with the length dependent on the device.

  SEQUENTIAL files may have FIXED or VARIABLE length records. If this entry is omitted for SEQUENTIAL files, VARIABLE-length records are assumed.

  If records are FIXED, the computer will pad each record on the right to ensure that it is the specified length. If the data is recorded in DISPLAY format, the computer will pad the record with spaces. If the INTERNAL format is used, the FIXED-length record will be padded with binary zeroes.
- *file-life* – Files you create with your TI Home Computer are considered PERMANENT, not temporary. You may omit this entry entirely, since the computer will assume a PERMANENT *file-life*.

Examples:

```
>100 OPEN #53:NAME$,FIXED,INT
 ERNAL,RELATIVE

 (computer assumes UPDATE)
```

```
>100 OPEN #11:NAME$,INPUT,INT
 ERNAL,SEQUENTIAL,VARIABLE 10
 0

>100 OPEN #75:"CS1",OUTPUT,FI
 XED

 (computer assumes SEQUENTIAL,
  DISPLAY,FIXED length of 64
  positions)
```

CLOSE

Cassette Recorder Information

Whenever the computer performs the CLOSE statement for a cassette tape device, you will receive instructions on your screen for operating the recorder, as shown on the right.

If you use the DELETE option with cassette recorders, no action beyond the closing of the file takes place.

Examples:

```
>NEW

>100 OPEN #24:"CS1",INTERNAL,
 INPUT,FIXED
>110 OPEN #19:"CS2",INTERNAL,
 OUTPUT,FIXED
 .
 . program lines
 .
>200 CLOSE #24
>210 CLOSE #19
>220 END
>RUN

 * REWIND CASSETTE TAPE   CS1
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY    CS1
   THEN PRESS ENTER


 * REWIND CASSETTE TAPE   CS2
   THEN PRESS ENTER

 * PRESS CASSETTE RECORD  CS2
   THEN PRESS ENTER
   .
   . program runs
   .


 * PRESS CASSETTE STOP    CS1
   THEN PRESS ENTER


 * PRESS CASSETTE STOP    CS2
   THEN PRESS ENTER

 ** DONE **
```

CLOSE

CLOSE #*file-number*[:DELETE]

The CLOSE statement "closes" or discontinues the association between a file and a program. After the CLOSE statement is performed, the "closed" file is not available to your program unless you OPEN it again. Also, the computer will no longer associate the closed file with the *file-number* you specified in the program. You can then assign that particular *file-number* to any file you wish.

If you use the DELETE option in the CLOSE statement, the action performed depends on the device used. As additional accessory devices become available, their accompanying manuals will describe the DELETE option.

If you attempt to CLOSE a file that you have not opened previously in your program, the computer will terminate your program with the "FILE ERROR" message.

In order to safeguard your files, the computer will automatically close any open files should an error occur which terminates your program. If a break occurs in your program, either by a BREAK command or your pressing the BREAK key (**SHIFT C**), open files are automatically closed *only* if one of the following occurs:

- you edit the program
- you terminate BASIC with the BYE command
- you RUN the program again
- you enter a NEW command

If you use the **SHIFT Q** key to leave your program, the computer will *NOT* close any open files and you could lose the data on these files. If you need to exit from your file-processing program before its normal end, follow these directions so that you won't lose any data:

- Press the "BREAK" key (**SHIFT C**) until the computer reacts with "BREAKPOINT AT xx." This may take several seconds.
- Enter BYE when the cursor reappears on the screen.

Examples:

```
>NEW

>100 OPEN #6:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED
>110 OPEN #25:"CS2",SEQUENTIA
 L,INTERNAL,OUTPUT,FIXED
 .
 . program lines
 .
>200 CLOSE #6:DELETE
>210 CLOSE #25
>220 END
```

DISPLAY-type Data

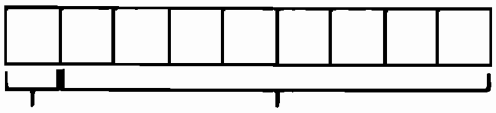
DISPLAY-type data has the same form as data entered from the keyboard. The computer knows the length of each data item in a DISPLAY-type record by the comma separators placed between items.

Each item in a DISPLAY-type record is checked to ensure that numeric values are placed in numeric variables as shown on the right in record 1. If the data-type doesn't match the variable-type, as in Record 2 on the right (JG is not a numeric value), an INPUT ERROR will occur and the program will terminate.

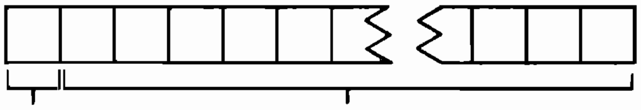
INTERNAL-type Data

INTERNAL-type data has the following form:

Numeric items:



String items:



The computer knows the length of each INTERNAL-type item by interpreting the one-position length indicator at the beginning of each item.

Limited validation of INTERNAL-type data-items is performed. All numeric items must be 9 positions long (8 digits plus one position which specifies the length) and must be valid representations of floating-point numbers. Otherwise, an INPUT ERROR will occur, and the program will terminate.

For FIXED-length INTERNAL records, reading beyond the actual data recorded in each record will cause padding characters (binary zeros) to be read. If you attempt to assign these characters to a numeric variable, an INPUT ERROR occurs. If strings are being read, a null string is assigned to the string variable.

Examples:

```
>NEW

>100 OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED 64
>110 INPUT #13:A,B,STATE$,D$,
 X,Y

 --INPUT RECORD 1=22,97.6,
     TEXAS,"AUTO LICENSE ",
     22000,-.07

 --INPUT RECORD 2=JG,22,TEXAS,
     PROPERTY TAX,42,15
```

INPUT

INPUT #*file-number*[.REC *numeric-expression*]:*variable-list*]

(See page 86 for a description of the INPUT form for use with the TI Home Computer keyboard.)

This form of the INPUT statement allows you to read data from an accessory device. The INPUT statement can be used only with files opened in INPUT or UPDATE mode. The *file-number* in the INPUT statement must be the *file-number* of a currently open file. *File-number* 0, the keyboard, may always be used. If you choose to use *file-number* 0, the INPUT statement is performed as described on page 86, except that you cannot specify an input-prompt.

The *variable-list* contains those variables which are assigned values when the INPUT statement is performed. Variable names in the *variable-list* are separated by commas and may be numeric and/or string variables.

Examples:

```
>NEW

>100 OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED
>110 INPUT #13:A,B,C$,D$,X,Y,
 Z$
>120 IF A=99 THEN 150
>130 PRINT A;B:C$:D$:X;Y:Z$
>140 GOTO 110
>150 CLOSE #13
>160 END
>RUN

 --data stored on tape will be
   printed on the screen

 ** DONE **
```

Filling the variable-list

When the computer reads records from a file, it stores each complete record internally in a temporary storage area called an input/output (I/O) buffer. A separate buffer is provided for each open *file-number*. Values are assigned to variables in the *variable-list* from left to right, using the data in this buffer. Whenever a *variable-list* has been filled with corresponding values, any data items left in the buffer are discarded unless the INPUT statement ends with a trailing comma. Using a trailing comma creates a "pending" input condition (see "Using Pending Inputs" on page 154).

```
>NEW

>100 OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED 64
>110 INPUT #13:A,B,C,D
 .
 . program lines
 .
>290 CLOSE #13
>300 END
>RUN

 --1st INPUT RECORD=22,77,56,
                    92

 --Results:
     A=22  B=77  C=56  D=92
```

If the *variable-list* in the INPUT statement is longer than the number of data items in the current record being processed, the computer will get the next record from the file and use its data items to complete the *variable-list*, as shown on the right.

When performing the INPUT statement, the computer will take different actions depending on whether the data stored is in DISPLAY or INTERNAL format.

```
>NEW

>100 OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED 64
>110 INPUT #13:A,B,C,D,E,F,G
 .
 . program lines
 .
>400 END

  --1ST INPUT RECORD=22,33.5
  --2ND INPUT RECORD=405,92
  --3RD INPUT RECORD=22,11023
  --4TH INPUT RECORD=99,100

  --Results:

    A=22  B=33.5  C=405  D=92

    E=-22 F=11023 G=99
```

Using Pending Inputs

A pending input condition is established when an INPUT statement with a trailing comma is performed. When the next INPUT statement using that file is encountered, one of the following actions will occur:

- If the next INPUT statement has no REC clause – the computer uses the data in the I/O buffer beginning where the previous INPUT statement stopped.
- If the next INPUT statement includes a REC clause – the computer terminates the pending input condition and reads the specified record into the file's I/O buffer.

If a pending input condition exists and a PRINT statement for the same file is performed, the pending input condition is terminated and the PRINT statement is performed as usual.

If you use a pending input with *file-number* 0, the error message "INCORRECT STATEMENT" is printed and the program stops running.

End-of-file

In sequential processing, to prevent an error when the computer has no more data to read, you will need to notify the computer that the end of the file has been reached. To make this easier for you, TI BASIC includes an End-of-File function called EOF (see page 156). Be sure to include the EOF statement immediately before the INPUT statement which reads a sequential file. In this way you can easily cause the computer to stop reading the input file when no more data is available. The usual procedure is to skip to a closing routine when EOF is reached.

Examples:

```
>NEW

>100 INPUT #0:A,B,
>110 PRINT A;B
>120 GOTO 100
>RUN
 ?
 * INCORRECT STATEMENT
   IN 100
```

```
>NEW

>100 OPEN #5:NAME$,SEQUENTIAL
 ,INTERNAL,INPUT,FIXED
>110 IF EOF(5) THEN 150
>120 INPUT #5:A,B
>130 PRINT A;B
>140 GOTO 110
>150 CLOSE #5
>160 END
```

Using INPUT with RELATIVE Files

(See page 146 for a description of RELATIVE file-organization.)

You may read RELATIVE files either sequentially or randomly. The computer sets up an internal counter to point to which record should be processed next. The first record in a file is record 0. Thus, the counter begins at zero and is incremented by +1 after each access to the file, either to read or to write a record. In the example on the right, the statements direct the computer to read the file sequentially.

The internal counter can be changed by using the REC clause. The *numeric-expression* following the keyword REC will be evaluated to designate a specific record number on the file. When the computer performs an INPUT statement with a REC clause, it reads the specified record from the designated file and places it in the I/O buffer. The REC clause can appear only in statements referencing RELATIVE files. The example on the right illustrates accessing a RELATIVE file randomly, using the REC clause.

Be sure to use the REC clause if you read and write records on the same file within a program. Since the same internal counter is incremented when records are either read or written for the same file, you may skip some records and write over others if REC is not used, as shown in the example on the right.

If the internal counter points to a record beyond the limits of the file when the computer tries to access the file, the program will terminate with an INPUT ERROR.

Examples:

```
>NEW

>100 OPEN #4:NAME$,RELATIVE,I
 NTERNAL,INPUT,FIXED 64
>110 INPUT #4:A,B,C$,D$,X
 .
 . program lines
 .
>200 CLOSE #4
>210 END
```

```
>NEW

>100 OPEN #6:NAME$,RELATIVE,I
 NTERNAL,UPDATE,FIXED 72
>110 INPUT K
>120 INPUT #6,REC K:A,B,C$,D$
 .
 . program lines
 .
>170 PRINT #6,REC K:A,B,C$,D$
 .
 . program lines
 .
>300 CLOSE #6
>310 END
```

```
>NEW

>100 OPEN #3:NAME$,RELATIVE,I
 NTERNAL,UPDATE,FIXED
>110 FOR I=1 TO 10
>120 INPUT #3:A$,B$,C$,X,Y
 .
 . program lines
 .
>230 PRINT #3:A$,B$,C$,X,Y
>240 NEXT I
>250 CLOSE #3
>260 END
>RUN

  --LINE 120-Reads records
              0,2,4,6,8...

  --LINE 130-Writes records
              1,3,5,7,9...
```

EOF—End-of-File Function

EOF (*numeric-expression*)

The end-of-file function determines if an end-of-file has been reached on a file stored on an accessory device. The argument specifies an open file-number (see page 145). The argument is the value obtained when the *numeric-expression* is evaluated. The normal rules for the evaluation of numeric expressions (page 40) are used here.

The value the function provides depends on the position of the file. The values supplied are:

Value	*Position*
0	Not end-of-file
+1	Logical end-of-file
−1	Physical end-of-file

A file is positioned at a logical end-of-file when all records on the file have been processed. A file is positioned at a physical end-of-file when no more space is available for the file.

This function and the example on the right cannot be used with cassette tape recorders. Its use with any other accessory devices will be more fully explained in their accompanying manuals.

Examples:

```
>NEW

>100 OPEN #2:NAME$,SEQUENTIAL
 ,INTERNAL,INPUT,FIXED
>110 IF EOF(2) THEN 160
>120 REM IF EOF GIVES ZERO
>130 INPUT #2:A,B,C
>140 PRINT A;B;C
>150 GOTO 110
>160 CLOSE #2
>170 END
```

The EOF function cannot be used with RELATIVE files or with some accessory devices. In these cases, you will need to create your own method for determining that the end-of-file has been reached.

One common end-of-file technique is to create a last record on the file that serves as an end-of-file indicator. It is called a "dummy" record because the data it contains is used only to mark the end of the file. For example, it could be filled with "9's." Whenever the computer inputs a record, you can check the data. If it is equal to "9's," then the computer has reached end-of-file and can skip to the closing routine.

The first example on the right creates a dummy record. In the next example, the computer checks for the dummy record as its end-of-file technique.

Cassette Recorder Information

- RELATIVE file-organization cannot be used with cassette devices.
- The EOF (End-of-File) function cannot be used with files on cassette recorders.
- You may specify a record length up to 192 positions (see page 148).
- Only cassette unit 1 (CS1) can be used for inputting data.

Examples:

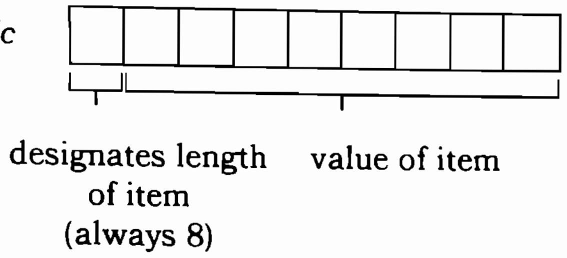
```
>NEW

>100 OPEN #2:"CS1",SEQUENTIAL
 ,FIXED,OUTPUT,INTERNAL
>110 READ A,B,C
>120 IF A=99 THEN 180
>130 E=A+B+C
>140 PRINT A;B;C;E
>150 PRINT #2:A,B,C,E
>160 GOTO 110
>170 DATA 5,10,15,10,20,30,10
 0,200,300,99,99,99
>180 PRINT #2:99,99,99,99
>190 CLOSE #2
>200 END
>RUN

 * REWIND CASSETTE TAPE    CS1
   THEN PRESS ENTER

 * PRESS CASSETTE RECORD   CS1
   THEN PRESS ENTER
  5  10  15  30
  10  20  30  60
  100  200  300  600

 * PRESS CASSETTE STOP     CS1
   THEN PRESS ENTER

 ** DONE **
```

```
>NEW

>100 OPEN #1:"CS1",INTERNAL,I
 NPUT,FIXED
>110 INPUT #1:A,B,C,E
>120 IF A=99 THEN 160
>130 F=A*E
>140 PRINT A;B;C;E;F
>150 GOTO 110
>160 CLOSE #1
>170 END
>RUN

 * REWIND CASSETTE TAPE    CS1
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY     CS1
   THEN PRESS ENTER
  5  10  15  30  150
  10  20  30  60  600
  100  200  300  600  6000

 * PRESS CASSETTE STOP     CS1
   THEN PRESS ENTER

 ** DONE **
```

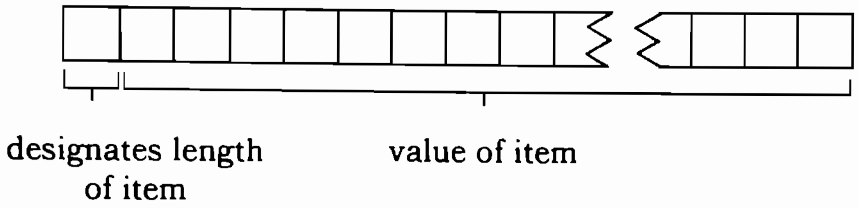
PRINT

When items in the *print-list* are written on the accessory storage device in INTERNAL format, they have the following characteristics:

Numeric items:



String items:



In the example on the right, the total length of the data recorded in INTERNAL format is 71 positions. Each numeric variable uses 9 positions. A$ is 18 characters long (line 110) plus 1 position to record the length of the string. B$ is 15 characters (line 120) plus 1. If the values of A$ and B$ change during the program, their lengths will vary according to whatever value is present when the record is written onto the files. In designing your record, therefore, become familiar with the data each variable might contain and plan your record to allow for the largest length possible.

Whenever you specify FIXED-length records, the computer will pad each INTERNAL-type record with binary zeros, if necessary, to bring each record to the specified length.

The computer will not allow a record to be longer than the specified or default length for the device you are using. If including all data in a *print-list* would cause this condition to occur for an INTERNAL-type record, the program will terminate with the message "FILE ERROR IN xx."

Examples:

```
>NEW

>100 OPEN #5:"CS1",SEQUENTIAL
 ,INTERNAL,OUTPUT,FIXED 128
>110 A$="TEXAS INSTRUMENTS "
>120 B$="HOME COMPUTER "
>130 READ X,Y,Z
>140 IF X=99 THEN 190
>150 A=X*Y*Z
>160 PRINT #5:A$,X,Y,Z,B$,A
>170 GOTO 130
>180 DATA 5,6,7,1,2,3,10,20,3
 0,20,40,60,1.5,2.3,7.6,99,99
 ,99
>190 CLOSE #5
>200 END
>RUN

 * REWIND CASSETTE TAPE    CS1
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY     CS1
   THEN PRESS ENTER

 --data written on tape


 * PRESS CASSETTE STOP     CS1
   THEN PRESS ENTER

 ** DONE **
```

PRINT

PRINT #*file-number*[,REC *numeric-expression*][:*print-list*]

(See page 93 for a description of the PRINT format for printing on the computer screen.)

This form of the PRINT statement allows you to write data onto an accessory device. The PRINT statement can be used to write only on files opened in OUTPUT, UPDATE, or APPEND mode (see page 147). The *file-number* must be the *file-number* of a currently open file.

When the computer performs a PRINT statement, it stores the data in a temporary storage area called an input/output (I/O) buffer. A separate buffer is provided for each open *file-number*. If the PRINT statement does not end with a print-separator (comma, semicolon, or colon), the record is immediately written onto the file from the I/O buffer. If the PRINT statement ends with a print-separator, the data is held in the buffer and a "pending" print condition occurs (see "Using Pending Prints" on page 161).

The information you need for creating a *print-list* to record data on accessory file storage devices is discussed here. The *print-list* needed to display print lines (on a printer, etc.) is the same as the *print-list* described on page 93. You may use either DISPLAY or INTERNAL format for data stored on accessory devices. However, since these files are read only by the computer, by far the easiest-to-use and most efficient data-type is INTERNAL.

Using PRINT with INTERNAL-type Data

The *print-list* consists of numeric and string expressions separated by commas, colons, or semicolons. All print-separators in a *print-list* have the same effect for INTERNAL-type data – they only separate the items from each other and do not indicate spacing character positions in a record.

Examples:

```
>NEW

>100 OPEN #5:"CS1",SEQUENTIAL
 ,INTERNAL,OUTPUT,FIXED
 .
 . program lines
 .
>170 PRINT #5:A,B,C$,D$
 .
 . program lines
 .
>200 CLOSE #5
>210 END
```

```
>NEW

>100 OPEN #6:"CS2",SEQUENTIAL
 ,DISPLAY,OUTPUT,FIXED
 .
 . program lines
 .
>170 PRINT #6:A;",";B;",";C$;
 ",";D$
 .
 . program lines
 .
>200 CLOSE #6
>210 END
```

PRINT

Using PRINT with RELATIVE Files

(See page 146 for a description of RELATIVE file-organization.)

RELATIVE file records can be processed randomly or in sequence. The computer sets up an internal counter to point to which record should be processed next. The first record in a file is record 0. Thus, the counter begins at zero and is incremented by +1 after each file access, either to read or to write a record. In the example on the right, the PRINT statement directs the computer to write the file sequentially. It can later be processed either randomly or in sequence.

The internal counter can be changed by using the REC clause. The keyword REC must be followed by a *numeric-expression* whose value specifies in which position the record in the file is to be written. When the computer performs a PRINT statement with a REC clause, it begins building an output record in the I/O buffer. When this record is written onto the file, it will be placed at the location specified by the REC clause. You may use the REC clause only with RELATIVE files. The example on the right illustrates writing records randomly, using the REC clause.

Be sure to use the REC clause if you read and write records on the same file within a program. Since the same internal counter is incremented when records are either read or written for the same file, you could skip some records and write over others if REC is not used, as shown in the example on the right.

Examples:

```
>NEW

>100 OPEN #3:NAME$,RELATIVE,I
 NTERNAL,OUTPUT,FIXED 128
 .
 . program lines
 .
>150 PRINT #3:A$,B$,C$,X,Y,Z
 .
 . program lines
 .
>200 CLOSE #3
>210 END
```

```
>NEW

>100 OPEN #3:NAME$,RELATIVE,I
 NTERNAL,UPDATE,FIXED 128
>110 INPUT K
>120 INPUT #3,REC K:A$,B$,C$,
 X,Y,Z
 .
 . program lines
 .
>300 CLOSE #3
>310 END
```

```
>NEW

>100 OPEN #3:NAME$,RELATIVE,I
 NTERNAL,UPDATE,FIXED
>110 FOR I=1 TO 10
>120 INPUT #3:A$,B$,C$,X,Y
>130 PRINT #3:A$,B$,C$,X,Y
>140 NEXT I
>150 CLOSE #3
>160 END

  LINE 120-reads records 0,2,4,
                         6,8...

  LINE 130-writes records 1,3,
                          5,7,
                          9...
```

Using PRINT with DISPLAY-type Data on File Storage Devices

Although it is best to use INTERNAL format for data recorded on file storage devices which will be read by the computer, you may occasionally need to use DISPLAY-type records. Included here are several important considerations you must observe when using DISPLAY format.

- Records are created according to the specifications found in the PRINT statement of the INPUT-OUTPUT section (page 93).
- If including a data-item from the *print-list* would cause the record to be longer than the specified or default length for the device you are using, the item is not split but becomes the first item in the next record. If any single item is longer than the record length, the item will be split into as many records as required to store it. The program continues running normally and no warning is given.
- In order to later read DISPLAY-type files created with the PRINT statement, the data must look like it does when you enter it from the keyboard. Therefore, you must explicitly include the comma separators and quote marks needed by the INPUT statement when you write the record on the file. These punctuation marks are not automatically inserted when the PRINT statement is performed. They must be included as items in the *print-list*, as shown in line 170 on the right.
- Numeric items *do not* have a fixed length as they do in INTERNAL format. In DISPLAY-type files, the length of a numeric item is the same as it would be if it were displayed on the screen using the PRINT (see page 93) or DISPLAY (see page 98) statement (i.e., includes sign, decimal point, exponent, trailing space, etc.). For example, the number of positions required to print 1.35E−10 is ten.

Examples:

```
>NEW

>100 OPEN #10:"CS1",SEQUENTIA
 L,DISPLAY,OUTPUT,FIXED 128
 .
 . program lines
 .
>170 PRINT #10:"""";A$;""",";
 X;",";Y;",";Z;",""";B$;""","
 ;A
 .
 . program lines
 .
>300 CLOSE #10
>310 END
```

RESTORE

RESTORE #*file-number*[,REC *numeric-expression*]

(See page 92 for a description of the RESTORE statement used with the READ and DATA statements.)

The RESTORE statement repositions an open file at its beginning record (see the first example on the right), or at a specific record if the file is RELATIVE (see the second example on the right).

If the *file-number* specified in a RESTORE statement is not already open, the program will terminate with the message "FILE ERROR IN xx."

You may use the REC clause only with a RELATIVE file. The computer evaluates the *numeric-expression* following REC and uses the value as a pointer to a specific record on the file. If you RESTORE a RELATIVE file and do not use the REC clause, the file will be set to record 0.

If there is a pending PRINT record, the record will be written on the file before the RESTORE is performed. If there is a pending INPUT, the data in the I/O buffer is discarded.

RELATIVE files are not supported by cassette recorders.

Examples:

```
>NEW

>100 OPEN #2:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED 64
>110 INPUT #2:A,B,C$,D$,X
 .
 . program lines
 .
>400 RESTORE #2
>410 INPUT #2:A,B,C$,D$,X
 .
 . program lines
 .
>500 CLOSE #2
>510 END
```

```
>NEW

>100 OPEN #4:NAME$,RELATIVE,I
 NTERNAL,UPDATE,FIXED 128
>110 INPUT #4:A,B,C
 .
 . program lines
 .
>200 PRINT #4:A,B,C
 .
 . program lines
 .
>300 RESTORE #4,REC 10
>310 INPUT #4:A,B,C
 .
 . program lines
 .
>400 CLOSE #4
>410 END
```

Using Pending Prints

A record is always written onto a file whenever the computer performs a PRINT statement which has no trailing separator. A pending print condition is established when a PRINT statement with a trailing print-separator is performed. When the next PRINT statement using the file is encountered, one of the following actions occurs:

- If the next PRINT statement has no REC clause – the computer places the data in the I/O buffer immediately following the data already there.
- If the next PRINT statement has a REC clause – the computer writes the pending print record onto the file at the position indicated by the internal counter and performs the new PRINT-REC statement as usual.

If a pending print condition exists and an INPUT statement for the same file is encountered, the pending print record will be written onto the file at the position indicated by the internal counter, and the internal counter is incremented. Then the INPUT statement is performed as usual. If a pending print condition exists and the file is closed (see page 149) or restored (see page 162), the pending print record is written before the file is closed or restored.

Cassette Recorder Information

- You may specify any record length up to 192 positions (see page 148).
- You may process SEQUENTIAL files only (you cannot use RELATIVE file-organization with cassette tapes).

TABLE 2. CHARACTER 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 3. PATTERN-IDENTIFIER CONVERSION TABLE

Blocks	BINARY CODE (0 =off;1 =on)	HEXADECIMAL CODE
□□□□	0000	0
□□□■	0001	1
□□■□	0010	2
□□■■	0011	3
□■□□	0100	4
□■□■	0101	5
□■■□	0110	6
□■■■	0111	7
■□□□	1000	8
■□□■	1001	9
■□■□	1010	A
■□■■	1011	B
■■□□	1100	C
■■□■	1101	D
■■■□	1110	E
■■■■	1111	F

Appendix

TABLE 1. CHARACTER CODES

ASCII CODE	CHARACTER	ASCII CODE	CHARACTER
32	(Space)	64	@ (at sign)
33	! (exclamation point)	65	**A**
34	" (quote)	66	**B**
35	# (number or pound sign)	67	**C**
36	$ (dollar)	68	**D**
37	% (percent)	69	**E**
38	& (ampersand)	70	**F**
39	' (apostrophe)	71	**G**
40	((open parenthesis)	72	**H**
41	) (close parenthesis)	73	**I**
42	* (asterisk)	74	**J**
43	+ (plus)	75	**K**
44	, (comma)	76	**L**
45	- (minus)	77	**M**
46	. (period)	78	**N**
47	/ (slant)	79	**O**
48	**0**	80	**P**
49	**1**	81	**Q**
50	**2**	82	**R**
51	**3**	83	**S**
52	**4**	84	**T**
53	**5**	85	**U**
54	**6**	86	**V**
55	**7**	87	**W**
56	**8**	88	**X**
57	**9**	89	**Y**
58	: (colon)	90	**Z**
59	; (semicolon)	91	[(open bracket)
60	< (less than)	92	\ (reverse slant)
61	= (equals)	93	] (close bracket)
62	> (greater than)	94	` (exponentiation)
63	? (question mark)	95	_ (line)

TABLE 5. COLOR CODES

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

TABLE 6. HIGH-RESOLUTION COLOR COMBINATIONS

The following color combinations produce the sharpest, clearest character resolution on the TI-99/4 color monitor screen. Color codes are included in parentheses.

Black on Medium Green (2, 3)
Black on Light Green (2, 4)
Black on Light Blue (2, 6)
Black on Dark Red (2, 7)
Black on Cyan (2, 8)
Black on Medium Red (2, 9)
Black on Light Red (2, 10)
Black on Dark Yellow (2, 11)
Black on Light Yellow (2, 12)
Black on Dark Green (2, 13)
Black on Magenta (2, 14)
Black on Gray (2, 15)
Black on White (2, 16)
Medium Green on White (3, 16)
Light Green on Black (4, 2)
Light Green on White (4, 16)
Dark Blue on Light Blue (5, 6)
Dark Blue on Gray (5, 15)
Dark Blue on White (5, 16)
Light Blue on Gray (6, 15)
Light Blue on White (6, 16)
Dark Red on Light Yellow (7, 12)
Dark Red on White (7, 16)
Medium Red on Light Red (9, 10)
Medium Red on Light Yellow (9, 12)
Medium Red on White (9, 16)
Light Red on Black (10, 2)
Light Red on Dark Red (10, 7)
Dark Yellow on Black (11, 2)
Light Yellow on Black (12, 2)
Light Yellow on Dark Red (12, 7)
Dark Green on Light Green (13, 4)
Dark Green on Light Yellow (13, 12)
Dark Green on Gray (13, 15)
Dark Green on White (13, 16)
Magenta on Gray (14, 15)
Magenta on White (14, 16)
Gray on Black (15, 2)
Gray on Dark Blue (15, 5)
Gray on Dark Red (15, 7)
Gray on Dark Green (15, 13)
Gray on White (15, 16)
White on Black (16, 2)
White on Medium Green (16, 3)
White on Light Green (16, 4)
White on Dark Blue (16, 5)
White on Light Blue (16, 6)
White on Dark Red (16, 7)
White on Medium Red (16, 9)
White on Light Red (16, 10)
White on Dark Green (16, 13)
White on Magenta (16, 14)
White on Gray (16, 15)

SPLIT CONSOLE KEYBOARD

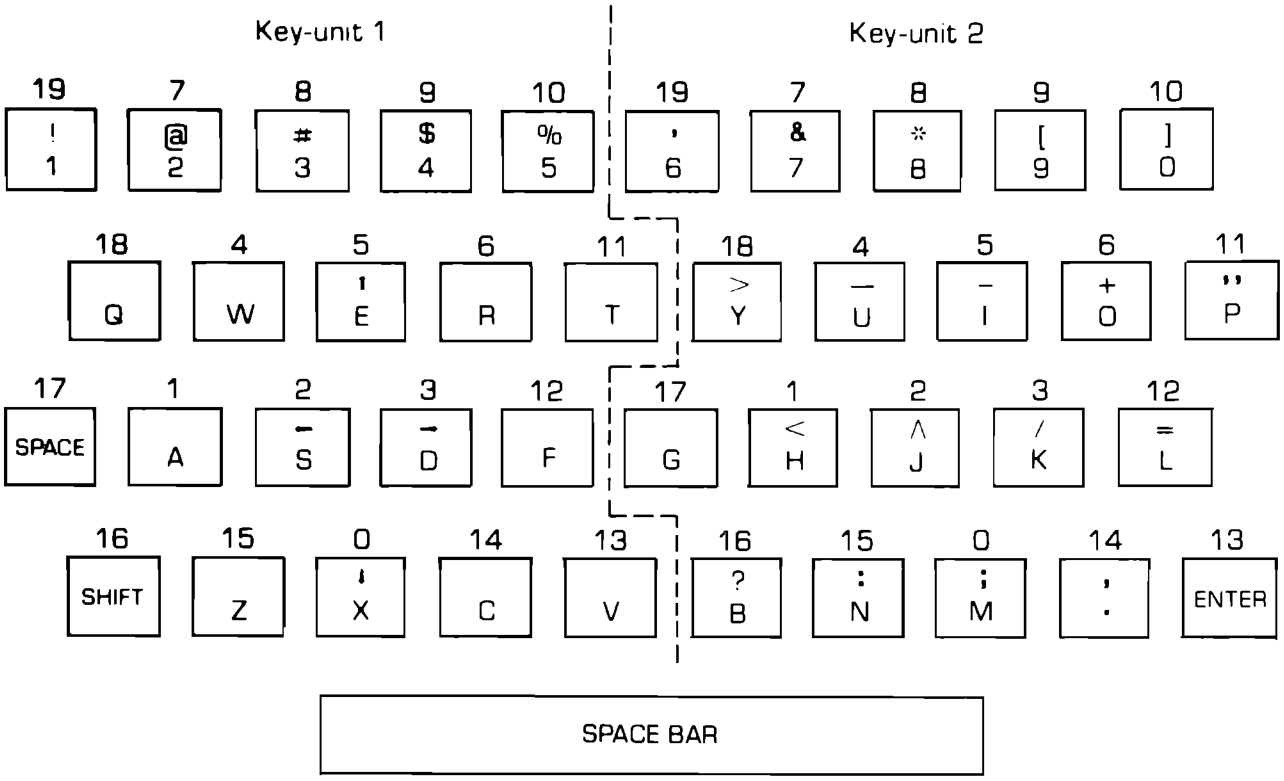


TABLE 4: CHARACTER CODES FOR SPLIT KEYBOARD

CODES	KEYS*	CODES	KEYS*
0	X,M	10	5,0
1	A,H	11	T,P
2	S,J	12	F,L
3	D,K	13	V,ENTER
4	W,U	14	C,.
5	E,I	15	Z,N
6	R,0	16	SHIFT,B
7	2,7	17	SPACE,G
8	3,8	18	Q,Y
9	4,9	19	1,6

*Note that the first key listed is on the left side of the keyboard, and the second key listed is on the right side of the keyboard.

Error Messages

I. Errors Found When Entering a Line

* BAD LINE NUMBER

1. Line number or line number referenced equals 0 or is greater than 32767
2. RESEQUENCE specifications generate a line number greater than 32767

* BAD NAME

1. The variable name has more than 15 characters

* CAN'T CONTINUE

1. CONTINUE was entered with no previous breakpoint or program was edited since a breakpoint was taken.

* CAN'T DO THAT

1. Attempting to use the following program statements as commands: DATA, DEF, FOR, GOTO, GOSUB, IF, INPUT, NEXT, ON, OPTION, RETURN
2. Attempting to use the following commands as program statements (entered with a line number): BYE, CONTINUE, EDIT, LIST, NEW, NUMBER, OLD, RUN, SAVE
3. Entering LIST, RUN, or SAVE with no program

* INCORRECT STATEMENT

1. Two variable names in a row with no valid separator between them (ABC A or A$A)
2. A numeric constant immediately follows a variable with no valid separator between them (N 257)
3. A quoted string has no closing quote mark
4. Invalid print separator between numbers in the LIST, NUMBER, or RESEQUENCE commands
5. Invalid characters following CONTINUE, LIST, NUMBER, RESEQUENCE, or RUN commands
6. Command keyword is not the first word in a line
7. Colon does not follow the device name in a LIST command

* LINE TOO LONG

1. The input line is too long for the input buffer

* MEMORY FULL

1. Entering an edit line which exceeds available memory
2. Adding a line to a program causes the program to exceed available memory

II. Errors Found When Symbol Table Is Generated

When RUN is entered but before any program lines are performed, the computer scans the program in order to establish a *symbol table*. A *symbol table* is an area of memory where the variables, arrays, functions, etc., for a program are stored. During this scanning process, the computer recognizes certain errors in the program, as listed below. The number of the line containing the error is printed as part of the message (for example: * BAD VALUE IN 100). Errors in this section are distinguished from those in section III, in that the screen color remains cyan until the symbol table is generated. Since no program lines have been performed at this point, all the values in the *symbol table* will be zero (for numbers) and null (for strings).

* BAD VALUE

1. A dimension for an array is greater than 32767
2. A dimension for an array is zero when OPTION BASE = 1

* CAN'T DO THAT

1. More than one OPTION BASE statement in your program
2. The OPTION BASE statement has a higher line number than an array definition

* FOR-NEXT ERROR

1. Mismatched number of FOR and NEXT statements

* INCORRECT STATEMENT

DEF

1. No closing ")" after a parameter in a DEF statement
2. Equals sign (=) missing in DEF statement
3. Parameter in DEF statement is not a valid variable name

TABLE 7. MUSICAL TONE FREQUENCIES

The following table gives frequencies (rounded to integers) of four octaves of the tempered scale (one half-step between notes). While this list does not represent the entire range of tones — or even of musical tones — it can be helpful for musical programming.

Frequency	*Note*	*Frequency*	*Note*
110	A	440	A (above middle C)
117	A#,B♭	466	A#,B♭
123	B	494	B
131	C (low C)	523	C (high C)
139	C#,D♭	554	C#,D♭
147	D	587	D
156	D#,E♭	622	D#,E♭
165	E	659	E
175	F	698	F
185	F#,G♭	740	F#,G♭
196	G	784	G
208	G#,B♭	831	G#,A♭
220	A (below middle C)	880	A (above high C)
220	A (below middle C)	880	A (above high C)
233	A#,B♭	932	A#,B♭
247	B	988	B
262	C (middle C)	1047	C
277	C#,D♭	1109	C#,D♭
294	D	1175	D
311	D#,E♭	1245	D#,E♭
330	E	1319	E
349	F	1397	F
370	F#,G♭	1480	F#,G♭
392	G	1568	G
415	G#,A♭	1661	G#,A♭
440	A (above middle C)	1760	A

OPEN, CLOSE, INPUT, PRINT, RESTORE

11. *File-number* negative or greater than 255
12. Number-of-records in the SEQUENTIAL option of the OPEN statement is non-numeric or greater than 32767
13. *Record-length* in the FIXED option of the OPEN statement is greater than 32767

POS

14. The *numeric-expression* in the POS statement is negative, zero, or larger than 32767

SCREEN

15. Screen *color-code* out of range

SEG$

16. The value of *numeric-expression1* (character position) or *numeric-expression2* (length of substring) is negative or larger than 32767

SOUND

17. *Duration, frequency, volume* or *noise* specification out of range

TAB

18. The value of the character position is greater than 32767 in the TAB function specification

* **CAN'T DO THAT**

1. RETURN with no previous GOSUB statement
2. NEXT with no previous matching FOR statement
3. The *control-variable* in the NEXT statement does not match the *control-variable* in the previous FOR statement
4. BREAK command with no line number

* **DATA ERROR**

1. No comma between items in DATA statement
2. *Variable-list* in READ statement not filled but no more DATA statements are available
3. READ statement with no DATA statement remaining
4. Assigning a string value to a numeric variable in a READ statement
5. *Line-number* in RESTORE statement is greater than the highest line number in the program

* **FILE ERROR**

1. Attempting to CLOSE, INPUT, PRINT, or RESTORE a file not currently open
2. Attempting to INPUT records from a file opened as OUTPUT or APPEND
3. Attempting to PRINT records on a file opened as INPUT
4. Attempting to OPEN a file which is already open

* **INCORRECT STATEMENT**

General

1. Opening "(", closing ")", or both missing
2. Comma missing
3. No line number where expected in a BREAK, UNBREAK, or RESTORE (BREAK 100,)
4. "+" or "−" not followed by a numeric expression
5. Expressions used with arithmetic operators are not numeric
6. Expressions used with relational operators are not the same type
7. Attempting to use a string expression as a subscript
8. Attempting to assign a value to a function
9. Reserved word out of order
10. Unexpected arithmetic or relational operator is present
11. Expected arithmetic or relational operator missing

Built-in Subprograms

12. In JOYST, the *x-return* and *y-return* are not numeric variables
13. In KEY, the *key-status* is not a numeric variable
14. In GCHAR, the third specification must be a numeric variable
15. More than three tone specifications or more than one noise specification in SOUND
16. CALL is not followed by a subprogram name

DIM

4. DIM statement has no dimensions or more than three dimensions
5. A dimension in a DIM statement is not a number
6. A dimension in a DIM statement is not followed by a comma or a closing ")"
7. The *array-name* in a DIM statement is not a valid variable name
8. The closing ")" is missing for array subscripts

OPTION BASE

9. OPTION not followed by BASE
10. OPTION BASE not followed by 0 or 1

* **MEMORY FULL**

1. Array size too large
2. Not enough memory to allocate a variable or function

* **NAME CONFLICT**

1. Assigning the same name to more than one array (DIM A(5), A(2,7))
2. Assigning the same name to an array and a simple variable
3. Assigning the same name to a variable and a function
4. References to an array have a different number of dimensions for the array (B=A(2,7)+2, PRINT A(5))

III. Errors Found When a Program Is Running

When a program is running, the computer may encounter statements that it cannot perform. An error message will be printed, and unless the error is only a warning the program will end. At that point, all variables in the program will have the values assigned when the error occurred. The number of the line containing the error will be printed as part of the message (for example: CAN'T DO THAT IN 210).

* **BAD ARGUMENT**

1. A built-in function has a bad argument
2. The string expression for the built-in functions ASC or VAL has a zero length (null string)
3. In the VAL function, the string expression is not a valid representation of a numeric constant

* **BAD LINE NUMBER**

1. Specified line number does not exist in ON, GOTO or GOSUB statement
2. Specified line number in BREAK or UNBREAK does not exist (warning only)

* **BAD NAME**

1. Subprogram name in a CALL statement is invalid

* **BAD SUBSCRIPT**

1. Subscript is not an integer
2. Subscript has a value greater than the specified or allowed dimensions of an array
3. Subscript 0 used when OPTION BASE 1 specified

* **BAD VALUE**

CHAR

1. *Character-code* out of range in CHAR statement
2. Invalid character in *pattern-identifier* in CHAR statement

CHR$

3. Argument negative or larger than 32767 in CHR$

COLOR

4. *Character-set-number* out of range in COLOR statement
5. *Foreground* or *background color code* out of range in COLOR statement

EXPONENTIATION (∧)

6. Attempting to raise a negative number to a fractional power

FOR

7. Step increment is zero in FOR-TO-STEP statement

HCHAR, VCHAR, GCHAR

8. *Row* or *column-number* out of range in HCHAR, VCHAR, or GCHAR statement

JOYST, KEY

9. *Key-unit* out of range in JOYST or KEY statement

ON

10. *Numeric-expression* indexing *line-number* is out of range

The second digit (Y) indicates what kind of error occurred.

Y Value	*Error Type*
0	Device name not found
3	Illegal operation
6	Device error

1. Invalid device or file name in DELETE, LIST, OLD, or SAVE command
2. Not enough memory to allocate an Input/Output buffer
3. This error can occur during file processing if an accessory device is accidentally disconnected while the program is running

- **MEMORY FULL**
  1. Not enough memory to allocate the specified character in CHAR statement
  2. GOSUB statement branches to its own *line-number*
  3. Program contains too many pending subroutine branches with no RETURN performed
  4. Program contains too many user-defined functions which refer to other user-defined functions
  5. Relational, string, or numeric expression too long
  6. User-defined function references itself
- **NUMBER TOO BIG** (warning given – value replaced by computer limit as shown below)
  1. A numeric operation produces an overflow (value greater than 9.9999999999999E127 or less than −9.9999999999999E127
  2. READing from DATA statement results in an overflow assignment to a numeric variable
  3. INPUT results in an overflow assignment to a numeric variable
- **STRING-NUMBER MISMATCH**
  1. A non-numeric argument specified for a built-in function, tab-function, or exponentiation operation
  2. A non-numeric value found in a specification requiring a numeric value
  3. A non-string value found in a specification requiring a string value
  4. Function argument and parameter disagree in type, or function type and expression type disagree for a user-defined function
  5. *File-number* not numeric in OPEN, CLOSE, INPUT, PRINT, RESTORE
  6. Attempting to assign a string to a numeric variable
  7. Attempting to assign a number to a string variable

File Processing-Input/Output Statements

17. Number sign (#) or colon (:) in *file-number* specification for OPEN, CLOSE, INPUT, PRINT, or RESTORE is missing
18. *File-name* in OPEN or DELETE must be a string expression
19. A keyword in the OPEN statement is invalid or appears more than once
20. The number of records in SEQUENTIAL option is less than zero in the OPEN statement
21. The record length in the FIXED option in the OPEN statement is less than zero or greater than 255
22. A colon (:) in the CLOSE statement is not followed by the keyword DELETE
23. *Print-separator* (comma, colon, semicolon) missing in the PRINT statement where required
24. *Input-prompt* is not a string expression in INPUT statement
25. *File-name* is not a valid string expression in SAVE, LOAD, or OLD command

General Program Statements

FOR

26. The keyword FOR is not followed by a numeric variable
27. In the FOR statement, the *control-variable* is not followed by an equals sign (=)
28. The keyword TO is missing in the FOR statement
29. In the FOR statement, the *limit* is not followed by the end of line or the keyword STEP

IF

30. The keyword THEN is missing or not followed by a line number

LET

31. Equals sign (=) missing in LET statement

NEXT

32. The keyword NEXT is not followed by *control-variable*

ON-GOTO, ON-GOSUB

33. ON is not followed by a valid numeric expression

RETURN

34. Unexpected word or character following the word RETURN

User-Defined Functions

35. The number of function arguments does not match the number of parameters for a user-defined function

- **INPUT ERROR**
  1. Input data is too long for Input/Output buffer (if data entered from keyboard, this is only a warning – data can be re-entered)
  2. Number of variables in the *variable-list* does not match number of data items input from keyboard or data file (warning only if from keyboard)
  3. Non-numeric data INPUT for a numeric variable. This condition could be caused by reading padding characters on a file record. (Warning only if from keyboard)
  4. Numeric INPUT data produces an overflow (warning only if from keyboard)

- **I/O ERROR** – This condition generates an accompanying error code as follows:

When an I/O error occurs, a two-digit error code (XY) is displayed with the message:

* I/O ERROR XY IN *line-number*

The first digit (X) indicates which I/O operation caused the error.

X Value	*Operation*
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

Applications Programs

Introduction

The programs in this section are designed to illustrate the use of many of the statements in TI BASIC. If you've never had any experience with programming, the best place to begin learning about TI BASIC is the *Beginner's BASIC* book included with your computer. When you've finished reading and working through the programs in that book, these programs will provide additional help in more complex programming. If you've had some experience in programming, these programs will provide a demonstration of many of the TI BASIC features.

The programs included here begin at a simple level and progressively become more complex. Thus, you can begin at whatever level you want. Most of the programs employ the color graphics and sound capabilities of the computer. These should provide you with a good basis for designing your own graphics and adding sound to your programs.

Accuracy Information

Displayed Results Versus Accuracy

Computers, like all other devices, must operate with a fixed set of rules within preset limits. The TI Home Computer uses especially powerful internal notation to represent numbers.

The mathematical tolerance of the computer is controlled by the number of digits it uses for calculations. The computer appears to use 10 digits as shown by the display, but actually uses more to perform all calculations. When rounded for display purposes, these extra digits help maintain the accuracy of the values presented. Example:

$\frac{1}{3} \times 3 = .9999999999$ (inaccurate)

The example shows that $\frac{1}{3} = .3333333333$, when multiplied by 3, produces an inaccurate answer. However, a 13-digit string of nines, when rounded to 10 places, will equal 1.0000000000.

The higher order mathematical functions use iterative and polynomial calculations. The cumulative rounding error is usually maintained below the 10-digit display so that no effect can be seen. The 13-digit representation of a number is three orders of magnitude from the displayed tenth digit. In this way the display assures that results are rounded accurately to ten digits.

Normally there is no need to even consider the undisplayed digits. On certain calculations, as with any computer, these digits may appear as an answer when not expected. The mathematical limits of a finite operation (word length, truncation and rounding errors) do not allow these digits to always be completely accurate. Therefore, when subtracting two expressions which are mathematically equal, the computer may display a nonzero result. Example:

```
X=⅔−⅓−⅓
PRINT X
1E−14
```

The final result indicates a discrepancy in the fourteenth digit.

The above fact is especially important when writing your own programs. When testing a calculated result to be equal to another value, precautions should be taken to prevent improper evaluation. For the above example, the statement X=1E−10*(INT(X*1E10)) will truncate the undisplayed digits of the variable X leaving only the rounded display value for further use.

Technical Information on Number Representation

Technically speaking, your computer uses a 7-digit Radix-100 mantissa for internal calculations. A single Radix-100 digit has a range of value from 0 to 99 in base-10 arithmetic. This means that a 7-digit Radix-100 number will correspond to decimal precision of 13 to 14 digits, depending on the value.

Radix-100 exponents range in value from −64 to +63 which yield decimal values of 10^{-128} to 10^{+126}. The Radix-100 mantissa and exponent combine to provide an equivalent decimal range of from −9.9999999999999E127 through −1.0000000000000E−128; zero; and then +1.0000000000000E−128 on through +9.9999999999999E127.

The internal format of each numerical value consists of eight bytes. The first byte contains the exponent and its sign, biased by 40 hex. The remaining bytes contain the mantissa, with the most significant digit first. The number is normalized so that the decimal point is immediately after the most significant digit. If the number is negative, then the first two bytes are complemented.

Examples:

1. The number 127_{10} is represented as:

EXP	MSD						LSD
41	01	1B	00	00	00	00	00

2. The fraction 0.5_{10} is represented as:

3F	32	00	00	00	00	00	00

3.a) The value of $\pi/2$ is represented as:

40	01	39	07	60	20	43	5F

b) The value of $-\pi/2$ is:

BF	FF	39	07	60	20	43	5F

Inchworm

This program creates an inchworm that moves back and forth across the screen. When the inchworm reaches the edge of the screen, an "uh-oh" sounds, and the inchworm turns around to go in the opposite direction.

These statements allow you to enter a color for the inchworm (color codes 2-3, 5-16 are recommended). The screen is then cleared. The CALL COLOR statement assigns the color you selected to character set 2. XDIR is used to designate which direction the inchworm moves (+1 indicates right and −1 indicates left).

This loop moves the inchworm across the screen. Line 180 computes where the next block is to be displayed and line 190 places the new block on the screen. The DELAY loop governs how fast the inchworm moves across the screen. Line 220 erases the old color block (so a continuous line won't be drawn) by placing a blank space over the block previously displayed at XOLD. Line 230 saves the current block position so a new one can then be computed. The loop is repeated until the inchworm reaches the edge of the screen.

Line 250 reverses the direction of the inchworm. Lines 260 and 270 produce the "uh-oh" sound. Then line 280 causes the loop to be performed again.

Examples:

```
>NEW

>100 REM INCHWORM
>110 CALL CLEAR
>120 INPUT "COLOR? ":C
>130 CALL CLEAR
>140 CALL COLOR(2,C,C)
>150 XOLD=1
>160 XDIR=1

>170 FOR I=1 TO 31
>180 XNEW=XOLD+XDIR
>190 CALL HCHAR(12,XNEW,42)
>200 FOR DELAY=1 TO 200
>210 NEXT DELAY
>220 CALL HCHAR(12,XOLD,32)
>230 XOLD=XNEW
>240 NEXT I

>250 XDIR=-XDIR
>260 CALL SOUND(100,392,2)
>270 CALL SOUND(100,330,2)
>280 GOTO 170
>RUN

 --screen clears

 COLOR? 7

 --screen clears

 --inchworm moves back and
   forth across the screen

 (Press SHIFT C to stop
  the program)
```

Random Color Dots

This program places random color dots in random locations on the screen. In addition, a random sound is generated and played when the dot is placed on the screen.

The RANDOMIZE statement causes a different sequence of numbers to be generated each time the program is run. The CALL CLEAR statement clears the screen.

This loop assigns each color code (2 through 16) to a different character set (codes 2 through 16).

These statements generate a random musical frequency for the CALL SOUND statement. Statement 170 generates notes from the tempered (twelve-tone) scale.

These statements generate a random character in the range of 40 through 159 and a random row and column location. (The color of the dot depends on the character set of the randomly chosen character.)

These statements produce the sound and place the solid color dot on the screen. Then the program loops back to generate a new sound, color dot, and location.

Examples:

```
>NEW

>100 REM RANDOM COLOR DOTS
>110 RANDOMIZE
>120 CALL CLEAR

>130 FOR C=2 TO 16
>140 CALL COLOR(C,C,C)
>150 NEXT C

>160 N=INT(24*RND)+1
>170 Y=110*(2^(1/12))^N

>180 CHAR=INT(120*RND)+40
>190 ROW=INT(24*RND)+1
>200 COL=INT(32*RND)+1

>210 CALL SOUND(-500,Y,2)
>220 CALL HCHAR(ROW,COL,CHAR)

>230 GOTO 160
>RUN

 -- screen clears

 --random color dots appear
   on the screen at different
   locations

 (Press SHIFT C to stop
  the program)
```

Secret Number

This program is a secret number game. The object is to guess the randomly chosen number between 1.and an upper limit you input. For each guess, you enter two numbers: a low and a high guess. The computer will tell you if the secret number is less than, greater than, or between the two numbers you enter. When you think you know the number, enter the same value for both the low and high guesses.

The RANDOMIZE statement ensures a different sequence of numbers each time the program is run. MSG1$ and MSG2$ are repeatedly used in PRINT statements. The CALL CLEAR statement clears the screen.

The INPUT statement stops the program and waits for you to enter a limit. Then the secret number is generated, and the screen is cleared. N is used to keep track of the number of guesses you make.

This INPUT statement accepts your low and high guesses. If you enter the same number for both guesses and you guess the secret number, the program transfers to line 300. If the secret number is less than your low number, the program transfers to line 260. If the secret number is greater than your high number, the program transfers to line 380. If the secret number is between your two numbers or equal to *one* of your numbers, the program continues.

These statements print a message to tell you where the secret number is in relation to your guesses. Then the program transfers to line 180 to allow you to guess again. If you guessed the secret number, the computer tells you how many guesses you took.

Examples:

```
>NEW

>100 REM SECRET NUMBER
>110 RANDOMIZE
>120 MSG1$="SECRET NUMBER IS"

>130 MSG2$="YOUR TWO NUMBERS"

>140 CALL CLEAR


>150 INPUT "ENTER LIMIT? ":LI
 MIT
>160 SECRET=INT(LIMIT*RND)+1
>170 CALL CLEAR
>180 N=N+1


>190 INPUT "LOW,HIGH GUESSES:
  ":LOW,HIGH
>200 IF LOW<>HIGH THEN 220
>210 IF SECRET=LOW THEN 300
>220 IF SECRET<LOW THEN 260
>230 IF SECRET>HIGH THEN 280


>240 PRINT MSG1$&" BETWEEN":M
 SG2$
>250 GOTO 180
>260 PRINT MSG1$&" LESS THAN"
 :MSG2$
>270 GOTO 180
>280 PRINT MSG1$&" LARGER THA
 N":MSG2$
>290 GOTO 180
>300 PRINT "YOU GUESSED THE S
 ECRET"
>310 PRINT "NUMBER IN ";N;"TR
 IES"
```

Marquee

This program puts a marquee on the screen. The colors are produced randomly, and a tone sounds each time a color bar is placed on the screen.

Examples:

```
>NEW
```

These statements clear the screen and assign each character set (2 through 16) to a different color. The RANDOMIZE statement ensures that a different set of colors will be produced each time the program is run.

```
>100 REM MARQUEE
>110 RANDOMIZE
>120 CALL CLEAR
>130 FOR S=2 TO 16
>140 CALL COLOR(S,S,S)
>150 NEXT S
```

These statements produce a border for the marquee.

```
>160 CALL HCHAR(7,3,64,28)
>170 CALL HCHAR(16,3,64,28)
>180 CALL VCHAR(7,2,64,10)
>190 CALL VCHAR(7,31,64,10)
```

This loop places color bars on the screen moving from left to right (columns 3 through 30). Each time a bar is placed on the screen, a tone sounds. The negative duration allows the sound to be cut off and a new sound to begin each time the CALL SOUND statement is performed. The subroutine beginning at line 310 generates the random colors and tones.

```
>200 FOR A=3 TO 30
>210 GOSUB 310
>220 CALL VCHAR(8,A,C,4)
>230 CALL SOUND(-150,Y,2)
>240 NEXT A
```

This loop is the same as the loop in lines 200 through 240 except that the color bars are placed on the screen moving from the right to the left. These color bars are placed below those generated by the previous loop. When the loop is finished, the program transfers to line 200 to begin at the left again.

```
>250 FOR A=30 TO 3 STEP -1
>260 GOSUB 310
>270 CALL VCHAR(12,A,C,4)
>280 CALL SOUND(-150,Y,2)
>290 NEXT A
>300 GOTO 200
```

This subroutine generates a random character (thus also generating a random color) for the CALL VCHAR statements (lines 220, 270). The assignment statements in lines 320 and 330 generate a random tone. The RETURN statement transfers the program to the statement following the GOSUB (lines 210, 260).

```
>310 C=INT(120*RND)+40
>320 N=INT(24*RND)+1
>330 Y=220*(2^(1/12))^N
>340 RETURN
>RUN

 --screen clears

 --marquee appears

 (Press SHIFT C to stop
  the program)
```

Bouncing Ball

This program moves a ball and bounces it off the edges of the screen. Each time the ball hits any side, a tone sounds, and the ball is deflected. The following special character is used to define the ball.

								Block Codes
		X	X	X	X			3C
	X	X	X	X	X	X		7E
X	X	X	X	X	X	X	X	FF
X	X	X	X	X	X	X	X	FF
X	X	X	X	X	X	X	X	FF
X	X	X	X	X	X	X	X	FF
	X	X	X	X	X	X		7E
		X	X	X	X			3C

Examples:

```
>NEW
```

These statements clear the screen and define character 96 as the ball.

```
>100 REM BOUNCING BALL
>110 CALL CLEAR
>120 CALL CHAR(96,"3C7EFFFFFF
 FF7E3C")
```

These statements allow you to input the color of the ball and the screen background color. Note that defining the screen color by using character set 1, which includes character 32 (the blank space), gives definite limits for the screen edge. The screen is cleared when the colors have been entered.

```
>130 INPUT "BALL COLOR? ":C
>140 INPUT "SCREEN COLOR? ":S

>150 CALL CLEAR
>160 CALL COLOR(9,C,S)
>170 CALL COLOR(1,S,S)
```

These statements give the starting position for the ball and set the parameters which will control the X and Y direction.

```
>180 X=16
>190 Y=12
>200 XDIR=1
>210 YDIR=1
```

These statements compute the next ball position. The direction the ball moves depends on the current values of XDIR (+1 indicates right, −1 indicates left) and YDIR (+1 indicates up, −1 indicates down).

```
>220 X=X+XDIR
>230 Y=Y+YDIR
```

These statements test to see if the new ball position is still on the screen. If either the row (Y) or column (X) value is out of range, then the program transfers to line 310 (column out of range) or line 360 (row out of range) to change the ball direction.

```
>240 IF X<1 THEN 310
>250 IF X>32 THEN 310
>260 IF Y<1 THEN 360
>270 IF Y>24 THEN 360
```

Secret Number

These statements offer you the choice of playing again or stopping the program. If you enter any character other than Y, the program ends. If you wish to play again, the counter for the number of guesses is set to zero, and you are asked if you want to set a new limit. If you enter Y, the program transfers back to line 140. If you enter any other character, the program transfers to line 160 to generate a new secret number.

Here is a sample of the program run. (Of course, your secret numbers will be different from the one shown here.)

Examples:

```
>320 PRINT "WANT TO PLAY AGAI
 N?"
>330 INPUT "ENTER Y OR N: ":A
 $
>340 IF A$<>"Y" THEN 390
>350 N=0
>360 PRINT "WANT TO SET A NEW
  LIMIT?"
>370 INPUT "ENTER Y OR N: ":B
 $
>380 IF B$="Y" THEN 140 ELSE
 160
>390 END
```

```
>RUN

 --screen clears

 ENTER LIMIT? 20

 --screen clears

 LOW,HIGH GUESSES: 1,10
 SECRET NUMBER IS BETWEEN
 YOUR TWO NUMBERS

 LOW,HIGH GUESSES: 1,5
 SECRET NUMBER IS LARGER THAN
 YOUR TWO NUMBERS

 LOW,HIGH GUESSES: 7,7
 YOU GUESSED THE SECRET
 NUMBER IN  3 TRIES
 WANT TO PLAY AGAIN?
 ENTER Y OR N: N

 ** DONE **
```

Checkbook Balance

Once each month all of us have the opportunity to tackle "balancing" our checkbooks against our bank statements. Normally, the checkbook balance will not agree with the balance shown on the bank statement because there are checks and deposits that haven't cleared yet. This program will help you balance your checkbook quickly and easily.

Examples:

```
>NEW
```

These statements clear the screen and allow you to input the balance shown on your bank statement.

```
>100 REM CHECKBOOK BALANCE
>110 CALL CLEAR
>120 INPUT "BANK BALANCE? ":B
 ALANCE
```

These statements give instructions for entering your outstanding check numbers and amounts. Note that DISPLAY and PRINT can be used interchangeably.

```
>130 DISPLAY "ENTER EACH OUTS
 ANDING"
>140 DISPLAY "CHECK NUMBER AN
 D AMOUNT."
>150 DISPLAY
>160 DISPLAY "ENTER A ZERO FO
 R THE"
>170 DISPLAY "CHECK NUMBER WH
 EN FINISHED."
>180 DISPLAY
```

This loop sets up the procedure for entering each check number and amount. These values are stored in arrays. If the check number equals zero, the program transfers out of the loop. CTOTAL is the total amount of outstanding checks. Each time a check amount is input, the program transfers to line 190 to input another check number and amount.

```
>190 N=N+1
>200 INPUT "CHECK NUMBER? ":C
 NUM(N)
>210 IF CNUM(N)=0 THEN 250
>220 INPUT "CHECK AMOUNT? ":C
 AMT(N)
>230 CTOTAL=CTOTAL+CAMT(N)
>240 GOTO 190
```

These statements give instructions for entering your outstanding deposits.

```
>250 DISPLAY "ENTER EACH OUTS
 TANDING"
>260 DISPLAY "DEPOSIT AMOUNT.
 "
>270 DISPLAY
>280 DISPLAY "ENTER A ZERO AM
 OUNT"
>290 DISPLAY "WHEN FINISHED."

>300 DISPLAY
```

This loop asks for and accepts each outstanding deposit amount. If the deposit amount equals zero, the program transfers out of the loop. DTOTAL is the total amount of outstanding deposits. After each outstanding deposit is added to the total, the program transfers to line 310 to accept another deposit amount.

```
>310 M=M+1
>320 INPUT "DEPOSIT AMOUNT? "
 :DAMT(M)
>330 IF DAMT(M)=0 THEN 360
>340 DTOTAL=DTOTAL+DAMT(M)
>350 GOTO 310
```

If the new ball position is still on the screen, then the screen is cleared to erase the old ball location. The ball is then displayed at the new location designated by Y and X.

Examples:

```
>280 CALL CLEAR
>290 CALL HCHAR(Y,X,96)
>300 GOTO 220
```

These statements change the direction of the ball if X is out of range. The CALL SOUND statement produces the "bouncing" tone. Lines 330 and 340 check to see if Y is also out of range. If it is, the program transfers to change the Y direction. If not, the program transfers to line 220 to compute a new ball position.

```
>310 XDIR=-XDIR
>320 CALL SOUND(30,380,2)
>330 IF Y<1 THEN 360
>340 IF Y>24 THEN 360
>350 GOTO 220
```

These statements change the direction of the ball if Y is out of range. The CALL SOUND statement produces the "bouncing" tone. The program then transfers to line 220 to compute the new ball position.

```
>360 YDIR=-YDIR
>370 CALL SOUND(30,380,2)
>380 GOTO 220
>RUN

 --screen clears

 BALL COLOR? 5
 SCREEN COLOR? 15

 --ball appears in center of
   screen and begins bouncing

 (Press SHIFT C to stop
  the program)
```

Codebreaker

Examples:

Codebreaker is a game in which the computer generates a four-digit code number, and you try to guess it. Zeros are not allowed, and no two digits may be the same. Even with these restrictions, there are 3024 possible codes, making slim your chances of guessing the number on the first try. Your guess is automatically scored by the computer. Your score for each guess is displayed in the form "N.R," where N is the number of digits in your trial number that appear in the secret number and are positioned correctly and R is the number of digits in your guess which although correct, are improperly placed. For example, if the number generated by the computer is 8261 and you guess 6285, you receive a score of 1.2. This indicates that one number you guessed is in the right place (the 2) and that two of your other numbers (8 and 6) are present in the secret number, but not in the right place. A score of 4.0 indicates that your guess is correct.

```
>NEW
```

The RANDOMIZE statement ensures that a different number will be generated each time the program is run. After the screen is cleared, the computer generates the four-digit number. Note that each digit is stored separately in the array, N. The J-loop beginning at line 160 ensures that no two digits in the number generated are the same. The number of tries is set to zero for each new four-digit number generated.

```
>100 REM CODEBREAKER GAME
>110 RANDOMIZE
>120 CALL CLEAR
>130 FOR I=1 TO 4
>140 N(I)=INT(9*RND)+1
>150 IF I=1 THEN 190
>160 FOR J=1 TO I-1
>170 IF N(I)=N(J) THEN 140
>180 NEXT J
>190 NEXT I
>200 TRIES=0
```

The INPUT statement stops the program and waits for you to enter your guess. Be sure to enter a four-digit integer number. Each time you guess a number, the score is set to zero, and the number of tries is increased by one.

```
>210 INPUT "ENTER GUESS? ":GU
 ESS
>220 SCORE=0
>230 TRIES=TRIES+1
```

Line 250 takes the last digit from the guess so that it may be compared against the code number. If the digit matches the code number in the same position, then the score is increased by 1. If not, then the L-loop is used to compare the digit against the other positions in the code number. If it matches any other position in the code number, then .1 is added to the score. Line 340 eliminates the last digit from the guess, so that the next digit can be taken for the comparison. When all four digits have been compared, the program continues at line 360.

```
>240 FOR K=4 TO 1 STEP -1
>250 DIGIT=(GUESS/10-INT(GUES
 S/10))*10
>260 IF DIGIT<>N(K) THEN 290
>270 SCORE=SCORE+1
>280 GOTO 340
>290 FOR L=1 TO 4
>300 IF N(L)<>DIGIT THEN 330
>310 SCORE=SCORE+.1
>320 GOTO 340
>330 NEXT L
>340 GUESS=INT(GUESS/10)
>350 NEXT K
```

Checkbook Balance

These statements compute and display the new balance. Then you enter the current balance in your checkbook. (Be sure you have subtracted bank service charges before you enter the current balance.) The correction necessary to make your checkbook agree with the bank statement is then computed and displayed.

Examples:

```
>360 NBAL=BALANCE-CTOTAL+DTOT
 AL
>370 DISPLAY "NEW BALANCE= ";
 NBAL
>380 INPUT "CHECKBOOK BALANCE
 ? ":CBAL
>390 DISPLAY "CORRECTION= ";N
 BAL-CBAL
>400 END
```

Here is a sample program run.

```
>RUN

 --screen clears

 BANK BALANCE? 940.26

 ENTER EACH OUTSTANDING
 CHECK NUMBER AND AMOUNT.

 ENTER A ZERO FOR THE
 CHECK NUMBER WHEN FINISHED.

 CHECK NUMBER? 212
 CHECK AMOUNT? 76.83
 CHECK NUMBER? 213
 CHECK AMOUNT? 122.87
 CHECK NUMBER? 216
 CHECK AMOUNT? 219.50
 CHECK NUMBER? 218
 CHECK AMOUNT? 397.31
 CHECK NUMBER? 219
 CHECK AMOUNT? 231.00
 CHECK NUMBER? 220
 CHECK AMOUNT? 138.25
 CHECK NUMBER? 0
 ENTER EACH OUTSTANDING
 DEPOSIT AMOUNT.

 ENTER A ZERO AMOUNT
 WHEN FINISHED.

 DEPOSIT AMOUNT? 450
 DEPOSIT AMOUNT? 0
 NEW BALANCE=  204.5

 CHECKBOOK BALANCE? 209.15
 CORRECTION= -4.65

 ** DONE **
```

Character Definition

This program allows you to define special graphics characters using the computer. An 8×8 grid is displayed on the screen. You then choose which "dots" to turn *on* and which to leave turned *off*. After the character has been designed, the program determines and displays the HEX string to be entered in the CALL CHAR statement.

These statements define the *off* dot character (line 120) and the *on* dot character (line 130). Black is used as the foreground color (*on* dot) and white is used as the background color (*off* dot). The screen is then cleared and the labels needed on the screen are displayed at the necessary locations. Note that the subroutine beginning at line 770 is used to print a string horizontally on the screen and the subroutine beginning at line 820 is used to print a string vertically on the screen. The R-loop is used to place the 8×8 grid (all dots turned *off*) on the screen.

This loop allows you to turn the "dots" either *on* or *off*. To turn a dot *on*, press the 1 key. To leave a dot turned *off*, press the 0 key. The cursor starts in the upper left corner (row 1, column 1) of the grid. Each time you press a key, the dot is turned *on* or *off* and the cursor moves to the next position. When the end of a row is reached, the cursor automatically moves to the next row. When the last "dot" is turned *on* or *off*, the program continues to determine the HEX string. Line 430 performs a logical OR. If the key you pressed was not a zero or a one, the program transfers back to line 370 to accept a new key input. Errors in the grid can be corrected before the last dot (row 8, column 8) is entered by using the LEFT arrow and RIGHT arrow keys. If either of these keys is pressed, then the program transfers to the subroutine beginning at line 870. The subroutine moves the cursor in the appropriate direction and to the next row up or down as necessary.

These statements determine the hexadecimal code for each row in the grid. When the code is determined, character 102 is defined to be the character shown on the large grid. The newly defined character is then displayed on the screen at row 8, column 20. The character is also displayed in a 3-by-3 pattern. Then the hexadecimal code defining that character is displayed. Lines 630 through 720 print instructions on the screen for you to define a new character. If you are finished defining characters, press Q and the program stops. If you press any other key, the program transfers to line 140 to clear the screen and begin again.

Examples:

```
>NEW

>100 REM CHARACTER DEFINITION

>110 DIM B(8,8)
>120 CALL CHAR(100,"")
>130 CALL CHAR(101,"FFFFFFFF
 FFFFFFF")
>140 CALL COLOR(9,2,16)
>150 CALL CLEAR
>160 M$="AUTO CHARACTER DEFIN
 ITION"
>170 Y=3
>180 X=4
>190 GOSUB 770
>200 M$="12345678"
>210 Y=8
>220 GOSUB 770
>230 GOSUB 820
>240 M$="0=OFF=WHITE"
>250 Y=22
>260 X=4
>270 GOSUB 770
>280 M$="1=ON=BLACK"
>290 Y=23
>300 GOSUB 770
>310 FOR R=1 TO 8
>320 CALL HCHAR(8+R,5,100,8)
>330 NEXT R
>340 FOR R=1 TO 8
>350 FOR C=1 TO 8
>360 CALL HCHAR(8+R,4+C,30)
>370 CALL KEY(0,KEY,STATUS)
>380 IF STATUS=0 THEN 370
>390 IF (KEY<>8)+(KEY<>9)=-2
 THEN 420
>400 GOSUB 870
>410 GOTO 360
>420 KEY=KEY-48
>430 IF (KEY<0)+(KEY>1)<=-1 T
 HEN 370
>440 B(R,C)=KEY
>450 CALL HCHAR(8+R,4+C,100+K
 EY)
>460 NEXT C
>470 NEXT R

>480 HEX$="0123456789ABCDEF"
>490 M$=""
>500 FOR R=1 TO 8
>510 LOW=B(R,5)*8+B(R,2)*4+B(
 R,7)*2+B(R,8)+1
>520 HIGH=B(R,1)*8+B(R,2)*4+B
 (R,3)*2+B(R,4)+1
>530 M$=M$&SEG$(HEX$,HIGH,1)&
 SEG$(HEX$,LOW,1)
>540 NEXT R
>550 CALL CHAR(102,M$)
>560 CALL HCHAR(8,20,102)
>570 FOR R=0 TO 2
>580 CALL HCHAR(12+R,20,102,3
 )
>590 NEXT R
```

These statements print the score for each guess. Strings are used in displaying the score to insure that the score is always displayed in the "N.R " format. If the score is an integer number, then a ".0" (line 370) must be added after the number. If the score is less than one, then a "0" (line 400) must be added before the number. If the score is a non-integer and greater than one, then just the score itself is printed (line 420). If the score is not equal to 4, the program transfers to line 210 to accept another guess.

Examples:

```
>360 IF INT(SCORE)<>SCORE THE
 N 390
>370 PRINT STR$(SCORE)&".0"
>380 GOTO 430
>390 IF SCORE>1 THEN 420
>400 PRINT "0"&STR$(SCORE)
>410 GOTO 430
>420 PRINT STR$(SCORE)
>430 IF SCORE<>4 THEN 210
```

These statements print the number of tries you took to guess the code number. Then the computer asks if you want to play again. If you enter Y, the program transfers to line 110 to generate a new number. If you enter anything else, the program stops.

```
>440 PRINT "YOU TOOK "&STR$(T
 RIES)&" TRIES TO GUESS"
>450 PRINT "THE CODE NUMBER."

>460 DISPLAY "WOULD YOU LIKE
 TO PLAY AGAIN"
>470 INPUT "ENTER Y OR N: ":A
$
>480 IF A$="Y" THEN 110
>490 END
```

Here is a sample of a program run. (Of course, your code numbers will be different.)

```
>RUN

 --screen clears

 ENTER GUESS? 1234
 0.1
 ENTER GUESS? 5678
 2.1
 ENTER GUESS? 9238
 1.0
 ENTER GUESS? 5694
 1.0
 ENTER GUESS? 5198
 2.1
 ENTER GUESS? 5718
 4.0
 YOU TOOK 6 TRIES TO GUESS
 THE CODE NUMBER.
 WOULD YOU LIKE TO PLAY AGAIN
 ENTER Y OR N: N

 ** DONE **
```

Graphics Match

This game program gives an example of developing special graphics for your own use. There are six different graphics characters defined. These are: heart, cherry, bell, lemon, diamond, and bar. To play the game you need only to run the program. The computer generates three random numbers in the range 1 through 6. Each time a number is generated, the picture corresponding to the number is displayed on the screen. Scoring depends on how many and in what way the three pictures match. When the three pictures and the score have been displayed, you are offered the choice of playing again.

Examples:

```
>NEW
```

These statements define the colors for each of the characters. The colors used are:

Graphics Character	*Color*
Heart	Medium Red
Cherry	Medium Red with Dark Green stem
Bell	Light Blue with Black handle
Lemon	Dark Yellow
Diamond	Dark Green
Bar	Dark Blue

```
>100 REM GRAPHICS MATCH
>110 CALL COLOR(9,7,16)
>120 CALL COLOR(10,13,16)
>130 CALL COLOR(11,2,16)
>140 CALL COLOR(12,6,16)
>150 CALL COLOR(13,11,16)
>160 CALL COLOR(14,5,16)
```

A white background is used for all of the pictures.

These statements define the heart.

```
>170 CALL CHAR(96,"00001C3E7F
 7F7F7F")
>180 CALL CHAR(97,"0000387CFE
 FEFEFE")
>190 CALL CHAR(98,"3F1F0F0703
 01")
>200 CALL CHAR(99,"FCF8F0E0C0
 80")
```

Block Codes																	*Block Codes*
00																	00
00																	00
1C				X	X	X					X	X	X				38
3E			X	X	X	X	X			X	X	X	X	X			7C
7F		X	X	X	X	X	X	X	X	X	X	X	X	X	X		FE
7F		X	X	X	X	X	X	X	X	X	X	X	X	X	X		FE
7F		X	X	X	X	X	X	X	X	X	X	X	X	X	X		FE
7F		X	X	X	X	X	X	X	X	X	X	X	X	X	X		FE
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
1F				X	X	X	X	X	X	X	X	X	X				F8
0F					X	X	X	X	X	X	X	X					F0
07						X	X	X	X	X	X						E0
03							X	X	X	X							C0
01								X	X								80
00																	00
00																	00

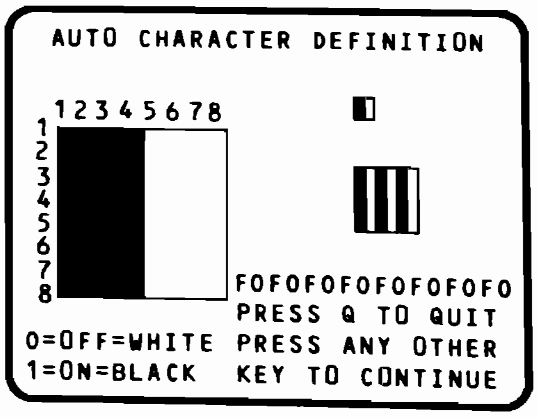
These subroutines print a given string beginning at a specified row and column on the screen. Lines 770 through 810 print a string horizontally. Lines 820 through 860 print a string vertically.

This subroutine is used to allow you to change the dots you have turned *on* or *off*. First, the new cursor location is checked. If the cursor is at the end of the line and the RIGHT arrow key is pressed, the cursor moves to the left side of the next line down. If the cursor is at the beginning of the line and the LEFT arrow key is pressed, the cursor moves to the right side of the next line up. If the cursor is at the upper left corner and the LEFT arrow key is pressed, the cursor moves to the lower right corner. If the cursor is at the lower right corner and the RIGHT arrow key is pressed, the cursor moves to the upper left hand corner.

A sample of the screen for a program run is shown at the right.

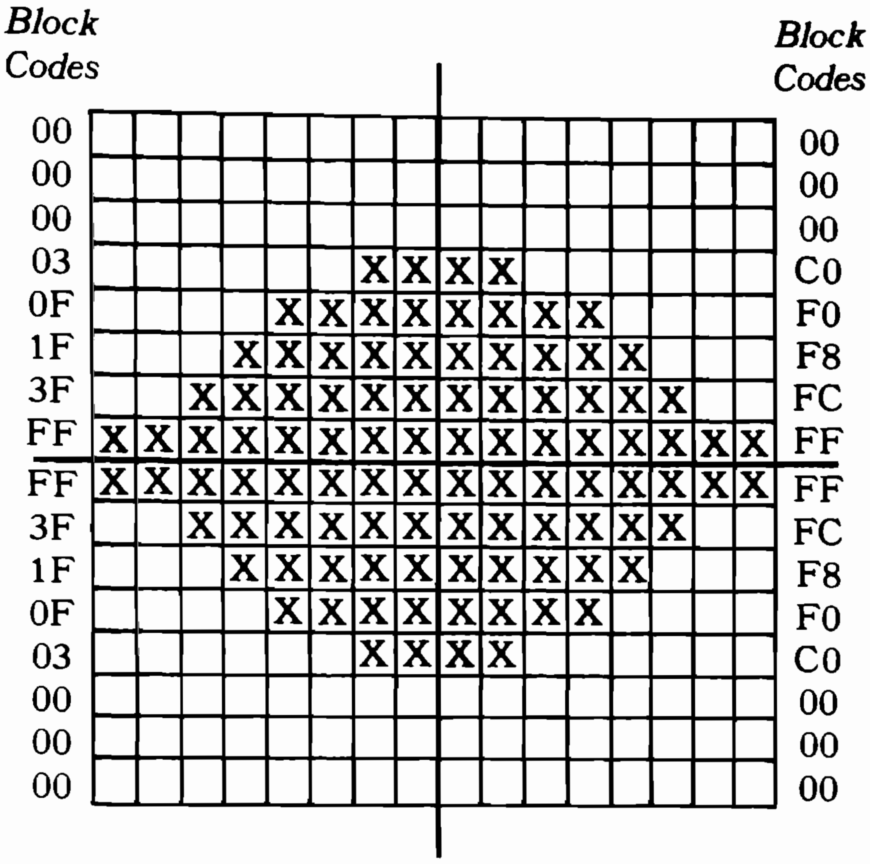
Examples:

```
>600 Y=16
>610 X=12
>620 GOSUB 770
>630 M$="PRESS Q TO QUIT"
>640 Y=18
>650 X=12
>660 GOSUB 770
>670 M$="PRESS ANY OTHER"
>680 Y=19
>690 GOSUB 770
>700 M$="KEY TO CONTINUE"
>710 Y=20
>720 GOSUB 770
>730 CALL KEY(0,KEY,STATUS)
>740 IF STATUS=0 THEN 730
>750 IF KEY<>81 THEN 140
>760 STOP
>770 FOR I=1 TO LEN(M$)
>780 CODE=ASC(SEG$(M$,I,1))
>790 CALL HCHAR(Y,X+I,CODE)
>800 NEXT I
>810 RETURN
>820 FOR I=1 TO LEN(M$)
>830 CODE=ASC(SEG$(M$,I,1))
>840 CALL HCHAR(Y+I,X,CODE)
>850 NEXT I
>860 RETURN
>870 CALL HCHAR(8+R,4+C,100+B
 (R,C))
>880 IF KEY=9 THEN 960
>890 C=C-1
>900 IF C<>0 THEN 1020
>910 C=8
>920 R=R-1
>930 IF R<>0 THEN 1020
>940 R=8
>950 GOTO 1020
>960 C=C+1
>970 IF C<>9 THEN 1020
>980 C=1
>990 R=R+1
>1000 IF R<>9 THEN 1020
>1010 R=1
>1020 RETURN
>RUN

 --screen clears
```



```
 AUTO CHARACTER DEFINITION

 12345678
1
2
3
4
5
6
7
8          F0F0F0F0F0F0F0F0
           PRESS Q TO QUIT
0=OFF=WHITE PRESS ANY OTHER
1=ON=BLACK  KEY TO CONTINUE
```

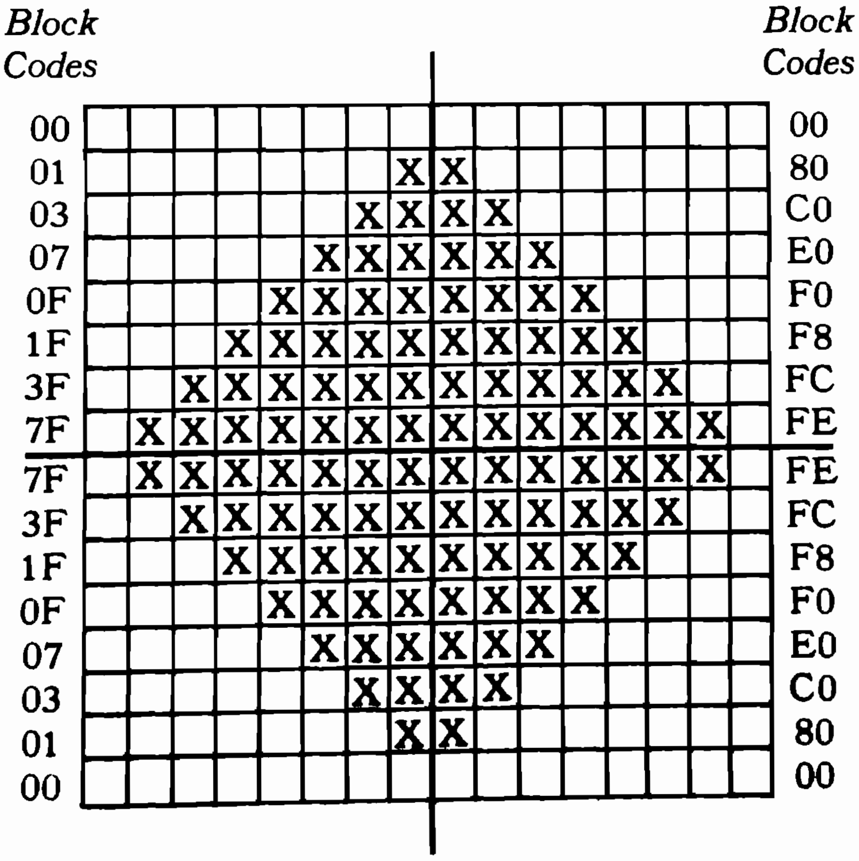
These statements define the lemon.



Examples:

```
>290 CALL CHAR(128,"000000030
 F1F3FFF")
>300 CALL CHAR(129,"000000C0F
 0F8FCFF")
>310 CALL CHAR(130,"FF3F1F0F0
 3")
>320 CALL CHAR(131,"FFFCF8F0C
 0")
```

These statements define the diamond.



```
>330 CALL CHAR(105,"000103070
 F1F3F7F")
>340 CALL CHAR(106,"0080C0E0F
 0F8FCFE")
>350 CALL CHAR(107,"7F3F1F0F0
 70301")
>360 CALL CHAR(108,"FEFCF8F0E
 0C080")
```

Note that in lines 190 and 200, the last four zeros are omitted. This saves time in entering the lines since the computer automatically fills the remaining length of the string with zeros.

These statements define the cherry.

Block Codes																	Block Codes
00																	00
00																	00
00														X	X		06
00													X				08
00												X					10
1F				X	X	X	X	X			X						20
3F			X	X	X	X	X	X		X							40
7F		X	X	X	X	X	X	X	X								80
7F		X	X	X	X	X	X	X	X	X	X						E0
7F		X	X	X	X	X	X	X	X	X	X	X					F0
7F		X	X	X	X	X	X	X	X	X	X	X					F0
7F		X	X	X	X	X	X	X	X	X	X	X					F0
3F			X	X	X	X	X	X	X	X	X	X					F0
3F			X	X	X	X	X	X	X	X	X						E0
1F				X	X	X	X	X	X	X							C0
00																	00

These statements define the bell.

Block Codes																	Block Codes
00																	00
00																	00
01								X	X								80
01								X	X								80
01								X	X								80
01								X	X								80
01								X	X								80
01								X	X								80
03							X	X	X	X							C0
07						X	X	X	X	X	X						E0
07						X	X	X	X	X	X						E0
07						X	X	X	X	X	X						E0
07						X	X	X	X	X	X						E0
0F					X	X	X	X	X	X	X	X					E0
07						X	X	X	X	X	X						E0
01								X	X								80

Examples:

```
>210 CALL CHAR(100,"000000000
 01F3F7F")
>220 CALL CHAR(104,"000006081
 0204080")
>230 CALL CHAR(101,"7F7F7F7F3
 F3F1F")
>240 CALL CHAR(102,"E0F0F0F0F
 0E0C0")
```

```
>250 CALL CHAR(112,"000001010
 1010101")
>260 CALL CHAR(113,"000080808
 0808080")
>270 CALL CHAR(120,"030707070
 70F0701")
>280 CALL CHAR(121,"C0E0E0E0E
 0F0E080")
```

Graphics Match

These statements determine the score you receive, as outlined in the table below. The line number indicates the line to which the program transfers to award the points.

Match	*Points*	*Line Number*
All pictures alike	Win 75	700
First two pictures, a cherry, lemon, or bar	Win 40	550
First two pictures a heart, bell, or diamond	Win 10	650
First and last pictures alike	Win 10	650
No match or last two pictures alike	Lose 10	610

Examples:

```
>490 REM SCORING
>500 IF PIC(1)<>PIC(2) THEN 5
 20
>510 IF PIC(2)=PIC(3) THEN 70
 0 ELSE 540
>520 IF PIC(1)<>PIC(3) THEN 6
 10
>530 GOTO 650
>540 IF PIC(1)/2<>INT(PIC(1)/
 2) THEN 650
```

These statements add 40 points to the accumulated score. Three tones sound and a message is displayed on the screen to indicate you have won a bonus worth 40 points. The program then transfers to line 770 to display the total points accumulated.

```
>550 TOTAL=TOTAL+40
>560 CALL SOUND(100,440,2)
>570 CALL SOUND(100,660,2)
>580 CALL SOUND(100,550,2)
>590 PRINT "BONUS--40 POINTS"

>600 GOTO 770
```

In line 610, ten points are subtracted from the total score. A tone sounds and a message is displayed to indicate you have lost ten points. The program then transfers to line 770 to display the new score.

```
>610 TOTAL=TOTAL-10
>620 CALL SOUND(100,110,1)
>630 PRINT "LOSE 10 POINTS"
>640 GOTO 770
```

In these statements, ten points are added to the total score. To indicate that you have won ten points, two tones sound and a message is displayed. Then the program transfers to line 770 to display the new score.

```
>650 TOTAL=TOTAL+10
>660 CALL SOUND(100,660,2)
>670 CALL SOUND(100,770,2)
>680 PRINT "WIN 10 POINTS"
>690 GOTO 770
```

These statements add 75 points to the total score. Five tones sound and a message indicating that you have won the jackpot is displayed.

```
>700 TOTAL=TOTAL+75
>710 CALL SOUND(100,440,2)
>720 CALL SOUND(100,550,2)
>730 CALL SOUND(100,440,2)
>740 CALL SOUND(100,660,2)
>750 CALL SOUND(100,880,2)
>760 PRINT "JACKPOT!--75 POIN
 TS"
```

The PRINT statement in line 770 prints your current score. The other statements offer you the choice of playing again or stopping the program. The CALL KEY statement (line 800) accepts an answer without your having to press **ENTER**. Pressing the **Y** key instructs the program to transfer back to line 410 to generate three new pictures. Pressing any other key stops the program.

```
>770 PRINT "CURRENT TOTAL POI
 NTS: ";TOTAL
>780 PRINT "WANT TO PLAY AGAI
 N?"
>790 PRINT "PRESS Y FOR YES"
>800 CALL KEY(0,KEY,STATUS)
>810 IF STATUS=0 THEN 800
>820 IF KEY=89 THEN 410
>830 END
```

These statements define the bar.

Block Codes																	*Block Codes*
00																	00
00																	00
00																	00
00																	00
00																	00
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
3F			X	X	X	X	X	X	X	X	X	X	X	X			FC
00																	00
00																	00
00																	00
00																	00
00																	00

Examples:

```
>370 CALL CHAR(136,"000000000
 03F3F3F")
>380 CALL CHAR(137,"000000000
 0FCFCFC")
>390 CALL CHAR(138,"3F3F3F")
>400 CALL CHAR(139,"FCFCFC")
```

The RANDOMIZE statement insures that a different sequence of pictures is generated each time the program is run. The variable C indicates the starting column location for the next picture. The I-loop generates a random number between 1 and 6, inclusive. The ON-GOSUB statement (line 460) transfers the program to the appropriate subroutine to place the picture on the screen. The pictures are displayed according to the following values:

PIC(I)	*Picture*
1	Heart
2	Cherry
3	Bell
4	Lemon
5	Diamond
6	Bar

After the picture is placed on the screen, the program returns to the loop to generate a new number and picture. When three pictures are displayed, the program continues to score the results.

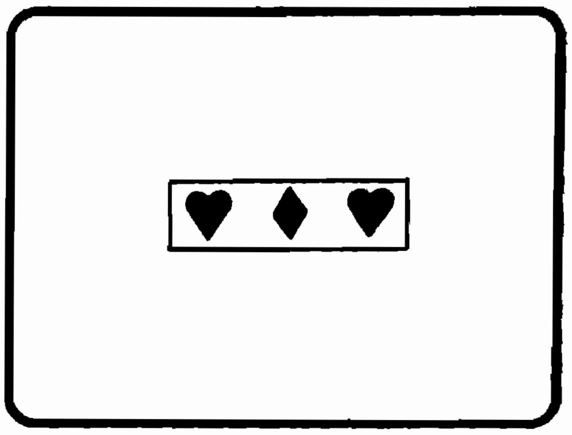
```
>410 RANDOMIZE
>420 CALL CLEAR
>430 C=14
>440 FOR I=1 TO 3
>450 PIC(I)=INT(6*RND)+1
>460 ON PIC(I) GOSUB 840,900,
 960,1020,1080,1140
>470 C=C+2
>480 NEXT I
```

Graphics Match

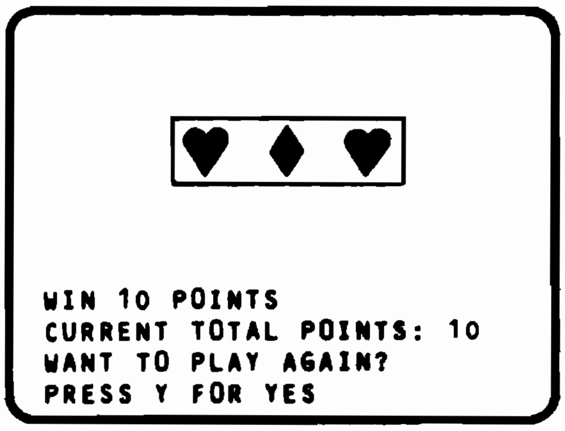
Here is a sample program run. Note that the computer screen remains cyan while the computer generates the symbol table and scans the program for errors (see page 168). This takes about a minute.

Examples:

```
>RUN
 --screen clears
```



```
--two tones sound
```



```
WIN 10 POINTS
CURRENT TOTAL POINTS: 10
WANT TO PLAY AGAIN?
PRESS Y FOR YES
```

```
PRESS Y FOR YES   N

 ** DONE **
```

Graphics Match

These six subroutines print each of the six pictures. The RETURN statements are used so that only one picture will be printed for each call to a subroutine.

Examples:

```
>840 REM PRINT HEART
>850 CALL HCHAR(12,C,96)
>860 CALL HCHAR(12,C+1,97)
>870 CALL HCHAR(13,C,98)
>880 CALL HCHAR(13,C+1,99)
>890 RETURN
>900 REM PRINT CHERRY
>910 CALL HCHAR(12,C,100)
>920 CALL HCHAR(12,C+1,104)
>930 CALL HCHAR(13,C,101)
>940 CALL HCHAR(13,C+1,102)
>950 RETURN
>960 REM PRINT BELL
>970 CALL HCHAR(12,C,112)
>980 CALL HCHAR(12,C+1,113)
>990 CALL HCHAR(13,C,120)
>1000 CALL HCHAR(13,C+1,121)
>1010 RETURN
>1020 REM PRINT LEMON
>1030 CALL HCHAR(12,C,128)
>1040 CALL HCHAR(12,C+1,129)
>1050 CALL HCHAR(13,C,130)
>1060 CALL HCHAR(13,C+1,131)
>1070 RETURN
>1080 REM PRINT DIAMOND
>1090 CALL HCHAR(12,C,105)
>1100 CALL HCHAR(12,C+1,106)
>1110 CALL HCHAR(13,C,107)
>1120 CALL HCHAR(13,C+1,108)
>1130 RETURN
>1140 REM PRINT BAR
>1150 CALL HCHAR(12,C,136)
>1160 CALL HCHAR(12,C+1,137)
>1170 CALL HCHAR(13,C,138)
>1180 CALL HCHAR(13,C+1,139)
>1190 RETURN
```

Glossary

Display – (noun) the home computer screen; (verb) to cause characters to appear on the screen.

Edit Mode – the mode used to change existing program lines. The EDIT mode is entered by using the Edit Command or by entering the line number followed by **SHIFT** [↓] or **SHIFT** [↑]. The line specified is displayed on the screen and changes can be made to any *character** using special keys described on page 66.

End-of-file – the condition indicating that all *data** has been read from a *file**.

Execute – to run a program; to perform the task specified by a *statement** or *command**.

Exponent – a number indicating the power to which a number or *expression** is to be raised; usually written at the right and above the number. For example, 2^8=2x2x2x2x2x2x2x2. In TI BASIC the exponent is entered following the ∧ symbol or following the letter "E" in *scientific notation**. For example, 2^8=2 ∧ 8; 1.3 × 10^{25}=1.3E25.

Expression – a combination of constants, variables, and operators which can be evaluated to a single result. Included are numeric, string, and relational expressions.

File – a collection of related data records stored on a device; also used interchangeably with *device** for input/output equipment which cannot use multiple files, such as a line printer.

Fixed-length records – records in a *file** which are all the same length. If a file has fixed-length records of 95 characters, each record will be allocated 95 *bytes** even if the *data** occupies only 76 positions. The computer will add padding characters on the right to ensure that the record has the specified length.

Function – a feature which allows you to specify as "single" operations a variety of procedures, each of which actually contains a number of steps; for example, a procedure to produce the square root via a simple reference name.

Graphics – visual constructions on the screen, such as graphs, patterns, and drawings, both stationary and animated. TI BASIC has built-in subprograms which provide easy-to-use color graphic capabilities.

Graphics line – a 32-character line used by the TI BASIC graphics subprograms.

Hardware – the various devices which comprise a computer system, including memory, the keyboard, the screen, disk drives, line printers, etc.

Hertz (Hz) – a unit of frequency. One Hertz=one cycle per second.

Hexadecimal – a base-16 number system using 16 symbols, 0-9 and A-F. It is used as a convenient "shorthand" way to express *binary** code. For example, 1010 in binary = A in hexadecimal, 11111111 = FF. Hexadecimal is used in constructing patterns for graphics characters in the CALL CHAR subprogram.

Immediate mode – see *Command Mode.*

Increment – a positive or negative value which consistently modifies a *variable**.

Input – (noun) *data** to be placed in computer memory; (verb) the process of transferring data into memory.

Input line – the amount of *data** which can be entered at one time. In TI BASIC, this is 112 characters.

Internal data-format – *data** in the form used directly by the computer. Internal numeric data is 8 *bytes** long plus 1 byte which specifies the length. The length for internal string data is one byte per character in the *string** plus one length-byte.

Integer – a whole number, either positive, negative, or zero.

I/O – Input/Output; usually refers to a device function. I/O is used for communication between the computer and other devices (e.g., keyboard, disk).

*See definition in *Glossary.*

Glossary

Accessory devices – additional equipment which attaches to the computer and extends its functions and capabilities. Included are preprogrammed *Command Modules** and units which send, receive or store computer data, such as printers and disks. These are often called peripherals.

Array – a collection of numeric or string variables, arranged in a list or matrix for processing by the computer. Each element in an array is referenced by a *subscript** describing its position in the list.

ASCII – the American Standard Code for Information Interchange, the code structure used internally in most personal computers to represent letters, numbers, and special characters. (See list on page 163.)

BASIC – an easy-to-use popular programming language used in most personal computers. The word BASIC is an acronym for "Beginners All-purpose Symbolic Instruction Code."

Baud – commonly used to refer to bits per second.

Binary – a number system based on two digits, 0 and 1. The internal language and operations of the computer are based on the binary system.

Branch – a departure from the sequential performance of program statements. An unconditional branch causes the computer to jump to a specified program line every time the branching statement is encountered. A conditional branch transfers program control based on the result of some arithmetic or logical operation.

Breakpoint – a point in the program specified by the BREAK command where program execution can be suspended. During a breakpoint, you can perform operations in the *Command Mode** to help you locate program errors. Program execution can be resumed with a CONTINUE command, unless editing took place while the program was stopped.

Buffer – an area of computer memory for temporary storage of an input or output record.

Bug – a hardware defect or programming error which causes the intended operation to be performed incorrectly.

Byte – a string of *binary** digits (bits) treated as a unit, often representing one data *character**. The computer's memory capacity is often expressed as the number of bytes available. For example, a computer with 16K bytes of memory has about 16,000 bytes available for storing programs and data.

Character – a letter, number, punctuation symbol, or special graphics symbol:

Command – an instruction which the computer performs immediately. Commands are not a part of a program and thus are entered with no preceding line number.

Command Mode – when no program is running, the computer is in the Command (or Immediate) Mode and performs each task as it is entered.

Command Modules – preprogrammed *ROM** modules which are easily inserted in the TI Home Computer to extend its capabilities.

Concatenation – linking two or more *strings** to make a longer string. The "&" is the concatenation operator.

Constant – a specific numeric or *string** value. A numeric constant is any real number, such as 1.2 or −9054. A string constant is any combination of up to 112 characters enclosed in quotes, such as "HELLO THERE" or "275 FIRST ST."

Cursor – a symbol which indicates where the next *character** will appear on the screen when you press a key.

Data – basic elements of information which are processed or produced by the computer.

Default – a standard characteristic or value which the computer assumes if certain specifications are omitted within a *statement** or a *program**.

Device (see *Accessory Devices*)

Disk – a mass storage device capable of random and sequential access.

*See definition in *Glossary*.

Reserved word – in programming languages, a special word with a predefined meaning. A reserved word must be spelled correctly, appear in the proper order in a *statement** or *command**, and cannot be used as a *variable** name.

ROM – read-only memory; certain instructions for the computer are permanently stored in ROM and can be accessed but cannot be changed. Turning the power off does not erase ROM.

Run Mode – when the computer is *executing** a program, it is in Run Mode. Run Mode is terminated when program execution ends normally or abnormally. You can cause the computer to leave Run Mode by pressing **SHIFT C** during program execution (see *Breakpoint**).

Scientific notation – a method of expressing very large or very small numbers by using a base number (*mantissa**) times ten raised to some power (*exponent**). To represent scientific notation in TI BASIC, enter the sign, then the mantissa, the letter E, and the power of ten (preceded by a minus sign if negative). For example, 3.264E4; −2.47E−17.

Scroll – to move the text on the screen so that additional information can be displayed.

Software – various programs which are executed by the computer, including programs built into the computer, *Command Module** programs, and programs entered by the user.

Statement – an instruction preceded by a line number in a program. IN TI BASIC, only one statement is allowed in a *program line**.

String – a series of letters, numbers, and symbols treated as a unit.

Subprogram – a predefined general-purpose procedure accessible to the user through the CALL statement in TI BASIC. Subprograms extend the capability of BASIC and cannot be easily programmed in BASIC.

Subroutine – a program segment which can be used more than once during the *execution** of a program, such as a complex set of calculations or a print routine. In TI BASIC, a subroutine is entered by a GOSUB statement and ends with a RETURN statement.

Subscript – a numeric expression which specifies a particular item in an *array**. In TI BASIC the subscript is written in parentheses immediately following the array name.

Trace – listing the order in which the computer performs program statements. Tracing the line numbers can help you find errors in a program flow.

Underflow – the condition which occurs when the computer generates a numeric value greater than −1E−128, less than 1E−128, and not zero. When an underflow occurs, the value is replaced by zero.

Variable – a name given to a value which may vary during program execution. You can think of a variable as a memory location where values can be replaced by new values during program execution.

Variable-length records – records in a *file** which vary in length depending on the amount of *data** per *record**. Using variable-length records conserves space on a file. Variable-length records can only be accessed sequentially.

*See definition in *Glossary*.

Glossary

Iteration – the technique of repeating a group of program statements; one repetition of such a group. See *Loop*.

Line – see *graphics line*, *input line*, *print line*, or *program line*.

Loop – a group of consecutive program lines which are repeatedly performed, usually a specified number of times.

Mantissa – the base number portion of a number expressed in *scientific notation**. In 3.264E+4, the *mantissa* is 3.264.

Mass storage device – an *accessory device**, such as a cassette recorder or disk drive, which stores programs and/or *data** for later use by the computer. This information is usually recorded in a format readable by the computer, not people.

Memory – see *RAM*, and *ROM*, and *mass storage device*.

Module – see *Command Module*.

Noise – various sounds which can be used to produce interesting sound effects. A *noise*, rather than a tone, is generated by the CALL SOUND *subprogram** when a negative frequency value is specified (−1 through −8).

Null string – a *string** which contains no characters and has zero length.

Number Mode – the mode assumed by the computer when it is automatically generating *program line** numbers for entering or changing statements.

Operator – a symbol used in calculations (numeric operators) or in relationship comparisons (relational operators). The numeric operators are +,−,*,/,∧. The relational operators are >,<,=,>=,<=,<>.

Overflow – the condition which occurs when a rounded value greater than 9.9999999999999E127 or less than −9.9999999999999E127 is entered or computed. When this happens, the value is replaced by the computer's limit, a warning is displayed, and the *program** continues.

Output – (noun) information supplied by the computer; (verb) the process of transferring information from the computer's memory onto a device, such as a screen, line printer, or *mass storage device**.

Parameter – any of a set of values that determine or affect the output of a *statement** or *function**.

Print line – a 28-position line used by the PRINT and DISPLAY statements.

Program – a set of statements which tell the computer how to perform a complete task.

Program line – a line containing a single *statement**. The maximum length of a program line is 112 *characters**.

Prompt – a symbol (>) which marks the beginning of each *command** or *program line** you enter; a symbol or phrase that requests input from the user.

Pseudo-random number – a number produced by a definite set of calculations (algorithm) but which is sufficiently random to be considered as such for some particular purpose. A true random number is obtained entirely by chance.

Radix-100 – a number system based on 100. See the information on number representation on page 173.

RAM – random access memory; the main memory where program statements and *data** are temporarily stored during program *execution**. New programs and data can be read in, accessed, and changed in RAM. Data stored in RAM is erased whenever the power is turned off or BASIC is exited.

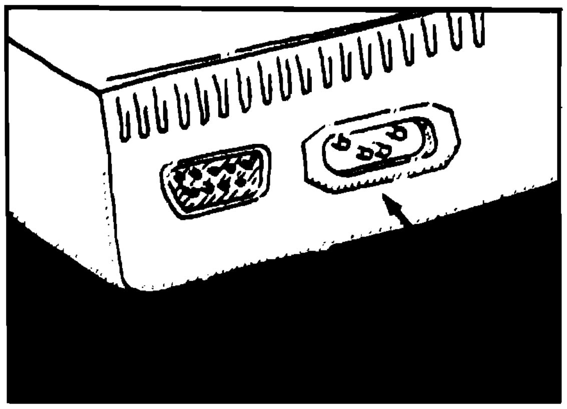
Record – (noun) a collection of related data elements, such as an individual's payroll information or a student's test scores. A group of similar records, such as a company's payroll records, is called a *file**.

*See definition in *Glossary*.

Monitor-Console Connection

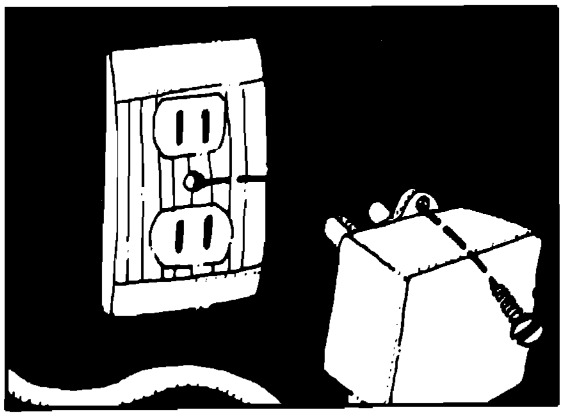
Connect Computer and Monitor Power Cords

Next, you'll connect the power cord (with transformer) to the computer. Connect the small 4-pin plug end into the outlet on the back of the computer as indicated below. *Notice that the pins only line up one way.*



Then, plug the power transformer into a regular wall outlet. It's best to plug the transformer into a wall outlet that is continuously "live," not one controlled by a wall switch.

You may want to secure the power transformer to the wall outlet. If so, simply remove the center screw on the wall plate, and then use it to attach the transformer through the hole provided. Follow the sketch below.



Finally, plug the monitor power cord into a continuously "live" wall outlet. (This plug has one regular and one wide blade, and is designed to fit into the wall outlet only one way. If the blades will not enter the outlet, try reversing the plug – see Monitor Manual for details.)

Check the Connections

Before you turn on your computer for the first time, follow these steps:

- Check to see that all connections between the computer console and the monitor are secure.
- Make sure both the computer and the monitor are "plugged in" to a live wall outlet.

Monitor-Console Connection

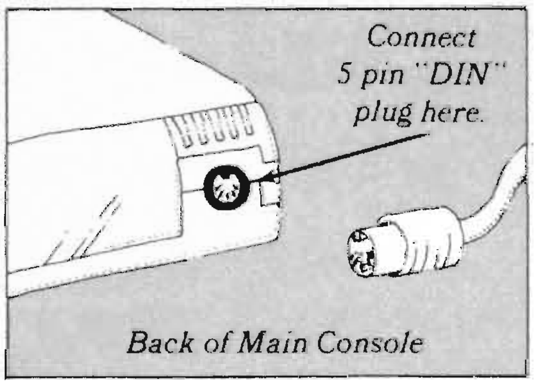
SET IT UP

First, select the right location for your computer system. Place the system on a non-metallic hard-topped surface (such as a table or desk) in a spot where the sunlight or bright light won't fall directly on the screen. Also, ventilation is necessary to keep your system running properly. Be sure air can flow freely through all the ventilation slots on the bottom, back, and top of the monitor and console.

Connect Console to Monitor

Connecting your TI-99/4 Home Computer to the display monitor requires only two simple steps. You'll use the cable packed with your monitor . Follow these steps:

1. Connect the single 5-pin plug (called a "DIN" plug) to your computer console at the point shown.

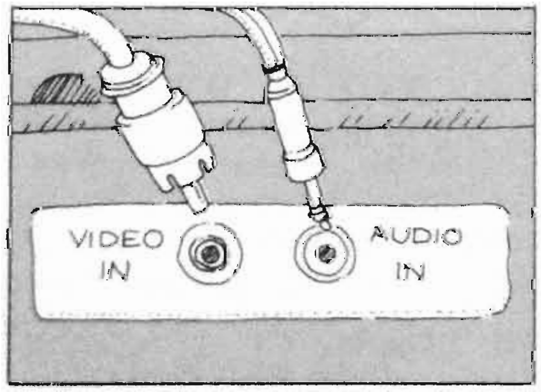


Back of Main Console

2. The other end of the cable (with two plugs) connects to your monitor.

   Connect the larger plug to the outlet labelled "video-in" on the back of your monitor as shown below.

   Connect the miniature plug to the outlet labelled "audio- in" on the back of your monitor as shown.



Note:
The video output of the computer is for direct connection to a video monitor only.

It is not designed for direct or indirect connection to a television receiver.

Maintenance and Service Information

When returning your Home Computer for repair or replacement, return the Home Computer, power cord, and any Command Modules which were involved when the difficulty occurred. For your protection, the Home Computer should be sent insured; Texas Instruments cannot assume any responsibility for loss of or damage to the Home Computer during shipment. It is recommended that the Home Computer be shipped in its original container to minimize the possibility of shipping damage. Otherwise, the Home Computer should be carefully packaged and adequately protected against shock and rough handling. Send shipments to the appropriate Texas Instruments Service Facility listed in the warranty. Please include information on the difficulty experienced with the Home Computer as well as return address information including name, address, city, state and zip code.

If you cannot determine whether the console or the TI Color Monitor has failed, both units must be returned.

If the Home Computer is in warranty, it will be repaired or replaced under the terms of the Limited Warranty. Out-of-warranty units in need of service will be repaired or replaced with reconditioned units (at TI's option), and service rates in effect at the time of return will be charged. Because our Service Facility serves the entire United States, it is not feasible to hold units while providing service estimates. For advance information concerning our flat-rate service charges, please call our toll-free number listed on page 203.

NOTE: The Color Monitor is too large to be sent via U.S. parcel post (fourth class mail) but may be sent via first class mail or by common carrier.

EXCHANGE CENTERS

If your Home Computer requires service, instead of returning the unit to a service facility for repair or replacement, you may elect to exchange the unit for a factory-reconditioned Home Computer of the same model (or equivalent model specified by TI) by going in person to one of the exchange centers which have been established across the United States. A handling fee will be charged by the exchange center for in-warranty exchanges of the Home Computer console and/or TI Color Monitor. Out-of-warranty exchanges will be charged at the rates in effect at the time of the exchange. Please refer to the enclosed Exchange Service listing or call the Consumer Relations Department for exchange fee information and the location of the nearest exchange center.

Maintenance and Service Information

IN CASE OF DIFFICULTY

In the event that you have difficulty with your Home Computer, the following instructions may help you to analyze the problem. You may be able to correct your Home Computer problem without returning it to a service facility. If the suggested remedies are not successful, contact the Consumer Relations Department by mail or telephone (refer to IF YOU HAVE QUESTIONS OR NEED ASSISTANCE later in this section). Please describe in detail the symptoms of your Home Computer.

If one of the following symptoms appears while operating with the optional peripheral(s) or accessories, remove the device. If the symptom disappears, refer to the manual for the peripheral or accessory in question.

SYMPTOM	*REMEDY*
Console indicator light will not come on when switch is turned on.	■ Check that transformer power cord is plugged into the wall. ■ Ensure that power cord is connected to the rear of the console.
No picture.	■ Check that power is on, and screen controls are set for optimum picture. Ensure that cables are properly connected as specified in the Color Monitor Operating Guide and Warranty.
No sound.	■ See that volume control is turned to proper level. Check connection of cables.
Cassette recorder will not operate when connected to console, but does work properly when not connected.	■ Ensure that cassette is connected to the 9-pin connector on the *rear* of the unit. (Not on left side.)
Cassette recorder will not Save or Load data properly.	■ See pages 17-18. ■ Remember that the cassette motor is controlled by the computer. Read instructions on pages 14-18 carefully.
Remote Controls will not operate.	■ Ensure that unit is connected to the 9-pin connector on the *left* side of the computer console. (Not on rear.) ■ Remember that only certain software is designed for use with the Remote Controls.
BASIC program or Equation Calculator is cleared by insertion of a Command Module.	■ This is a normal reset procedure designed to protect your color screen.
Stray characters appear or other erratic operation occurs or computer will not respond to keyboard input.	■ Static electricity discharges from the user to the console can alter program data stored in the internal memory. To correct this problem turn the console off and then on.

A Command Module especially designed to verify proper operation of the major functions of your system is available at your retailer. You can also purchase the *Diagnostic* module for use at home.

Index

If you have questions or need assistance

FOR GENERAL INFORMATION

If you have questions concerning Home Computer repair, or peripheral, accessory or software purchase, please call our Customer Relations Department at 800-858-4565 (toll free within the contiguous United States except Texas) or 800-692-4279 within Texas. The operators at these numbers cannot provide technical assistance.

FOR TECHNICAL ASSISTANCE

For technical questions such as programming, specific Home Computer applications, etc., you can call 806-741-2663. We regret that this is not a toll-free number, and we cannot accept collect calls.

As an alternative, you can write to:

Consumer Relations Department
Texas Instruments Incorporated
P.O. Box 53
Lubbock, Texas 79408

Because of the number of suggestions which come to Texas Instruments from many sources containing both new and old ideas, Texas Instruments will consider such suggestions only if they are freely given to Texas Instruments. It is the policy of Texas Instruments to refuse to receive any suggestions in confidence. Therefore, if you wish to share your suggestions with Texas Instruments, or if you wish us to review any BASIC language program which you have developed, please include the following statement in your letter:

> "All of the information forwarded herewith is presented to Texas Instruments on a nonconfidential, nonobligatory basis; no relationship, confidential or otherwise, expressed or implied, is established with Texas Instruments by this presentation. Texas Instruments may use, copyright, distribute, publish, reproduce, or dispose of the information in any way without compensation to me."

Index

Index

Three-Month Limited Warranty

THIS TEXAS INSTRUMENTS HOME COMPUTER CONSOLE WARRANTY EXTENDS TO THE ORIGINAL CONSUMER PURCHASER OF THE CONSOLE.

WARRANTY DURATION

This Home Computer console is warranted for a period of three (3) months from the date of the original purchase by the consumer.

WARRANTY COVERAGE

This Home Computer console is warranted against defective materials or workmanship. **THIS WARRANTY IS VOID IF THE CONSOLE HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.**

WARRANTY DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE HOME COMPUTER CONSOLE OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

WARRANTY PERFORMANCE

Please first contact the retailer from whom you purchased the console and determine the exchange policies of the retailer.

During the above three-month warranty period, your TI Home Computer console will be repaired or replaced with a new or reconditioned console of the same or equivalent model (at TI's option) when the console is returned either in person or by prepaid shipment to a Texas Instruments Service Facility listed below.

Texas Instruments strongly recommends that you insure the console for value, prior to shipment.

The repaired or replacement console will be warranted for three months from date of repair or replacement. Other than the cost of shipping the unit to Texas Instruments or postage, no charge will be made for the repair or replacement of in-warranty consoles.

Index

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U.S. Residents

Texas Instruments Service Facility
2303 North University
Lubbock, Texas 79415

Canadian Residents

Texas Instruments Service Facility
41 Shelley Road
Richmond Hill, Ontario, Canada
L4C5G4

Consumers in California and Oregon may contact the following Texas Instruments office for additional assistance or information.

Texas Instruments Consumer Service
831 South Douglas Street
El Segundo, California 90245
(213) 973-1803

Texas Instruments Consumer Service
10700 Southwest Beaverton Highway
Park Plaza West
Beaverton, Oregon 97005
(503) 643-6758

IMPORTANT NOTICE REGARDING PROGRAMS AND BOOK MATERIALS

The following should be read and understood *before* purchasing and/or using TI's Home Computer.

TI does not warrant that the programs contained in this computer and accompanying book materials will meet the specific requirements of the consumer, or that the programs and book materials will be free from error. The consumer assumes complete responsibility for any decision made or actions taken based on information obtained using these programs and book materials. Any statements made concerning the utility of TI's programs and book materials are not to be construed as express or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THESE PROGRAMS OR BOOK MATERIALS OR ANY PROGRAMS DERIVED THEREFROM AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THESE PROGRAMS OR BOOK MATERIALS, AND THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THIS HOME COMPUTER. MOREOVER, TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER AGAINST THE USER OF THESE PROGRAMS OR BOOK MATERIALS BY ANY OTHER PARTY.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

TEXAS INSTRUMENTS
INCORPORATED

Printed in U.S.A.