Texas Instruments
TI-99/4 Home Computer

TI Extended BASIC

FOR THE TI-99/4 HOME COMPUTER

A powerful, high-level programming language that expands the capability of your TI-99/4 Home Computer. Includes these features:

- *More than 40 new or expanded commands, statements, functions, and subprograms.*
- *Multiple-statement lines for speed and efficiency.*
- *Sprite (moving graphics) capability.*
- *Subprogram capability that lets you store commonly used subprograms on diskette for use as needed.*
- *The ability to load and run one program from another.*
- *Comprehensive program control of errors, warnings, and breakpoints.*
- *Direct screen control of input and output.*
- *Support for loading and running TMS9900 Assembly Language programs if the optional Memory Expansion unit (sold separately) is attached to the computer.*

*CONTENTS: TI Extended BASIC module
(36K bytes of preprogrammed memory)
Owner's reference manual*

IMPORTANT NOTICE REGARDING PROGRAMS AND BOOK MATERIALS

The following should be read and understood before purchasing and/or using TI Extended BASIC.

Texas Instruments does not warrant that the programs contained in the TI Extended BASIC module 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 MODULE. 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.

TI Extended BASIC

FOR THE TI-99/4 HOME COMPUTER

A powerful, high-level programming language that expands the capability of your TI-99/4 Home Computer. Includes these features:

- ***More than 40 new or expanded commands, statements, functions, and subprograms.***
- ***Multiple-statement lines for speed and efficiency.***
- ***Sprite (moving graphics) capability.***
- ***Subprogram capability that lets you store commonly used subprograms on diskette for use as needed.***
- ***The ability to load and run one program from another.***
- ***Comprehensive program control of errors, warnings, and breakpoints.***
- ***Direct screen control of input and output.***
- ***Support for loading and running TMS9900 Assembly Language programs if the optional Memory Expansion unit (sold separately) is attached to the computer.***

This book was developed and written by:

Robert E. Whitsitt, II
and other staff members of the Texas Instruments Learning Center
and the Texas Instruments Personal Computer Division.

With contributions by:

Tom M. Ferrio
Stanley R. Hume
Jacquelyn F. Quiram

Artwork and layout were coordinated and executed by:

Schenck Design Associates, Inc.

ISBN #0-89512-045-3
Library of Congress Catalog #80-54899

Table of Contents

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1

Introduction

INTRODUCTION

FEATURES

Texas Instruments Extended BASIC is a powerful computer programming language for use with the Texas Instruments TI-99/4 Home Computer. It has the features expected from a high level language plus additional features not available in many other languages, including those designed for use with large, expensive computers.

TI Extended BASIC goes beyond Texas Instruments BASIC to enhance the capability and flexibility of your computer system by adding these features:

- *Input and Output* — The ACCEPT statement allows the input of data from anywhere on the screen. You may clear the screen, accept only certain characters, and limit the number of characters entered using this statement. The DISPLAY statement has been enhanced to allow putting data anywhere on the screen, and DISPLAY ... USING, PRINT ... USING, and IMAGE have been added for ease in formatting data on the display screen and peripheral devices.
- *Subprograms* — Subprograms with local variables (affecting only values within the subprogram) can be written in TI Extended BASIC. Commonly used subprograms may be stored on a diskette and added to programs as needed. Statements included are SUB, SUBEND, and SUBEXIT. The MERGE command has been added and the SAVE command modified to allow the merging of programs from diskettes.
- *Sprites* — Sprites are specially defined graphics with the ability to move smoothly on the screen. To provide the sprite capability, the following subprograms have been included in TI Extended BASIC: COINC, DELSPRITE, DISTANCE, LOCATE, MAGNIFY, MOTION, PATTERN, POSITION, and SPRITE. COLOR and CHAR have been redesigned so they also can affect sprites.
- *Functions* — MAX, returning the larger of two numbers; MIN, returning the smaller of two numbers; and PI, returning the value of π, have been included in TI Extended BASIC.
- *Arrays* — Arrays may have up to seven dimensions instead of three.
- *String Handling* — The RPT$ function allows the repetition of a string.
- *Error Handling* — With TI Extended BASIC, you can choose what action is taken if there is a minor error (which in TI BASIC causes a warning message), a major error (which in TI BASIC causes an error message and stops the program), or a breakpoint (which in TI BASIC causes the program to halt). The new statements allowing this control are ON WARNING, ON ERROR, and ON BREAK. RETURN has been modified for use with error handling. The CALL ERR statement can be used to determine the nature of an error that occurs in a program.

- *RUN as a Statement* — RUN can be used as a statement as well as a command. RUN has also been modified to allow you to specify which program to run. As a result, one program can load and run another program from a diskette. You can, therefore, write programs of almost unlimited size by breaking them into pieces and letting each segment run the next.
- *Power-up Program Execution* — When TI Extended BASIC is first chosen, it searches for a program named LOAD on the diskette in disk drive 1. If that program exists, it is placed in memory and run.
- *Multiple Statement Lines* — TI Extended BASIC allows more than one statement to be on a line. This feature speeds program execution, saves memory, and allows logical units (for example FOR-NEXT loops) to be on a single line.
- *SAVE and LIST Protection* — You may protect your programs from being saved or listed, preventing unauthorized copies of and changes in your programs. This, in conjunction with the copy protection feature of the Disk Manager Module, can completely secure a TI Extended BASIC program.
- *IF-THEN-ELSE* — The IF-THEN-ELSE statement now allows statements as the consequences of the comparison. This expansion permits statements such as "IF X<4 THEN GOSUB 240 ELSE X=X+1".
- *Multiple Assignments* — TI Extended BASIC allows you to assign a value to more than one variable in a LET statement, saving statements and permitting more efficient programming.
- *Comments* — In addition to the REM statement, comments can be added to the ends of lines in TI Extended BASIC, allowing detailed internal documentation of programs.
- *Assembly Language Support* — With the optional Memory Expansion unit (available separately), TMS9900 assembly language subprograms may be loaded and run. The subprograms INIT, LOAD, LINK, and PEEK are used to access assembly language subprograms. There are no facilities for writing assembly language programs on the TI-99/4 Home Computer.
- *Information* — The SIZE command has been added to tell you how much memory remains unused in your computer. The VERSION subprogram returns a value which indicates the version of BASIC that is in use. The CHARPAT subprogram returns a character string indicating the pattern which defines a character.
- *Memory Expansion* — TI Extended BASIC allows the use of an optional Memory Expansion peripheral which permits much larger programs to be written.

CHANGES FROM TI BASIC

The enhancements described above have made some slight changes necessary in other areas of TI BASIC. Because of these, some programs written in TI-99/4 BASIC may not run in TI Extended BASIC.

- The maximum program size is now 864 bytes smaller than in TI BASIC. If you have the Memory Expansion peripheral, much larger programs may be written.
- The characters in character sets 15 and 16 are no longer available. That memory area is used by TI Extended BASIC to keep track of sprites.
- Most programs written in TI BASIC will also run in TI Extended BASIC without difficulty. Under certain circumstances, however, such as using a TI Extended BASIC keyword as a variable in a TI BASIC program, programs written in TI BASIC may not run in TI Extended BASIC. However, you can always load TI BASIC programs into TI Extended BASIC. Programs using the enhancements of TI Extended BASIC will not run correctly in TI BASIC.

HOW TO USE THIS MANUAL

This manual assumes that you are already experienced in programming with TI BASIC. Statements, commands, and functions that are the same as in TI BASIC are only discussed briefly here. For a complete discussion, see the *User's Reference Guide* that came with your TI-99/4 Home Computer.

The additional features of TI Extended BASIC are explained in detail and illustrated with examples and programs. To get the maximum use from TI Extended BASIC, read this manual carefully, entering and running the sample programs to see how they work. Even features that are unchanged from TI BASIC should be reviewed. You may find that you have been neglecting a useful statement or discover a new way to use statements in different combinations.

The remainder of this chapter reviews the basics of operating with TI Extended BASIC. The second chapter discusses the features of TI Extended BASIC and includes a detailed example of entering a program. The third chapter discusses the conventions of operation with TI Extended BASIC. The fourth chapter is a reference section which discusses, in alphabetical order, all TI Extended BASIC commands, statements, and functions.

The 14 appendices contain much useful information, including ASCII character codes, error codes, color codes, keyboard codes, and instructions on how to add suffixes to speech words.

HOW TO USE THE COMPUTER

Before using the computer with TI Extended BASIC, you must insert the *Solid State Software*™ Command Module into the computer. If the computer is off, slowly slide the module into the slot on the console until it is in place.

Then turn the computer on. (If you have peripherals, turn them on before turning on the computer.) The master title screen appears. If the computer is already on, return to the master title screen. Then slide the module into the slot.

Press any key to make the master selection list appear. The title of the module, TI EXTENDED BASIC, is third on the list. Type **3** to select TI Extended BASIC.

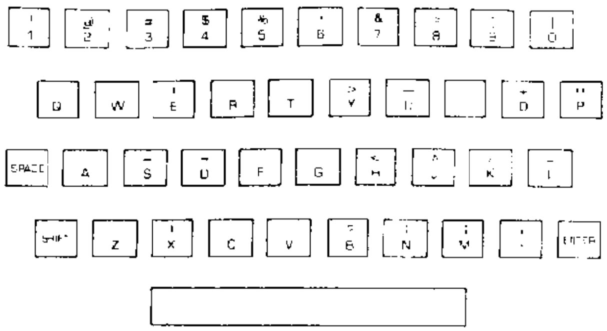
OPERATING IN TI EXTENDED BASIC

There are three main operating modes in TI Extended BASIC: Command Mode, Edit Mode, and Run Mode.

Command Mode is the mode entered when you choose TI Extended BASIC on the master selection list. In the Command Mode you may enter TI Extended BASIC commands, statements that may be used as commands, and program lines.

Edit Mode is used to edit existing lines of a TI Extended BASIC program. To enter Edit Mode, type a line number and press either **SHIFT E** (UP) or **SHIFT X** (DOWN). (TI BASIC also allows EDIT followed by a line number, which TI Extended BASIC does not allow.) The line specified is then displayed on the screen. You may change it by typing a new line, by typing over part of the old line, or by using the editing keys discussed below. You are also in the Edit Mode when you press **SHIFT R** (REDO) to repeat a program line or command.

In Run Mode, a TI Extended BASIC program is executed. You can stop a running program only by pressing **SHIFT C** (CLEAR), which causes a breakpoint, or with **SHIFT Q** (QUIT). *Note*: **SHIFT Q** (QUIT) also erases the entire program, returns you to the master title screen, and may delete information from some of your files. The use of BYE is recommended in place of **SHIFT Q** (QUIT) to leave TI Extended BASIC.



SPECIAL KEY FUNCTIONS

The following are the keys that have a special function when pressed at the same time as the **SHIFT** key: **E, D, S, X, R, T, G, F, C, Q**. Each of these keys is discussed below.

SHIFT E (UP) is used in the Edit Mode. If you are not in the Edit Mode, you may enter it by typing a line number and then pressing **SHIFT E** (UP). The line specified is then displayed on the screen and may be edited. If you are already in the Edit Mode, pressing **SHIFT E** (UP) enters the present line as you have changed it and displays the next lower numbered line in the program. Pressing **SHIFT E** (UP) when you are at the lowest numbered line in the program returns you to the Command Mode. If you are entering a line in the Command Mode, **SHIFT E** (UP) has the same effect as **ENTER**.

SHIFT D (RIGHT) moves the cursor one space to the right. The cursor does not erase or change the characters as it passes over them. At the end of a line on the screen, the cursor wraps around to the next screen line. When the cursor is at the end of an input line, it does not move.

SHIFT S (LEFT) moves the cursor one space to the left. The cursor does not erase or change characters as it passes over them. If the cursor is at the beginning of a line, the cursor does not move. If the cursor is at the left margin but not at the beginning of an input line, the cursor goes to the right margin of the screen line above it.

SHIFT X (DOWN) is used in the Edit Mode. If you are not in the Edit Mode, you may enter it by typing a line number and then pressing **SHIFT X** (DOWN). The line specified by the line number is then displayed on the screen and may be edited. If you are in the Edit Mode, pressing **SHIFT X** (DOWN) enters the present line as you have changed it and displays the next higher numbered line in the program. Pressing **SHIFT X** (DOWN) when you are at the highest numbered line in the program returns you to the Command Mode. If you are entering a line in the Command Mode, **SHIFT X** (DOWN) has the same effect as **ENTER**.

SHIFT R (REDO) causes the characters on the line previously input to reappear on the screen. Thus if you wish to enter a line similar to the most recently entered line, press **SHIFT R** (REDO). If you enter a line and make a mistake, you can recall the line using **SHIFT R** (REDO) and correct it using the Edit Mode features. This key lets you avoid retyping a long line.

SHIFT T (ERASE) erases all characters on the current line, but leaves the cursor on that line. If you are in the Command Mode, the cursor returns to the left margin of the screen and you may enter a new line, including the line number. However, if you are editing a line or the computer is providing the line numbers (through the use of NUM), the line number is not erased.

SHIFT G (INSERT) instructs the computer to accept inserted characters. Each subsequent key that you type is inserted at the cursor position and the character at the cursor position and all characters to the right of the cursor are shifted one position to the right. Insertion continues with each character typed until **ENTER** or one of the other special function keys is pressed. Characters at the end of a long input line may be lost.

SHIFT F (DELETE) deletes the character that the cursor is on and shifts all characters to the right of the cursor one position to the left.

SHIFT C (CLEAR) performs different functions depending on the mode that you are in. If you are in the Edit Mode, any changes that were made to the line are ignored, including **SHIFT T** (ERASE), and the computer returns to Command Mode. If you are in Run Mode, the program is stopped with a breakpoint. If you are in Command Mode, the characters that you have typed on the current line are deleted. When using **SHIFT C** (CLEAR) to stop a program, hold the keys down until TI Extended BASIC recognizes the breakpoint.

SHIFT Q (QUIT) returns the computer to the master title screen. *When you press* **SHIFT Q** *(QUIT), all data and program material are erased from the computer's memory. If you are using a disk system, some of your data files may be lost.* Leave TI Extended BASIC by entering BYE instead of using **SHIFT Q** (QUIT).

ENTER indicates that you have finished typing the information on the current line and are ready for the computer to process it.

CHAPTER
2

Overview of TI Extended BASIC

This chapter briefly describes the TI Extended BASIC commands, statements, and functions and suggests ways in which you can use them. The first eight sections are Commands; Assignments and Input; Output; Functions, Subroutines, and Subprograms; Sound, Speech, and Color; Sprites; Debugging; and Error Handling. The final section is an example of the entry of a program showing the entry process and the use of some of the TI Extended BASIC elements.

COMMANDS

Commands tell the computer to perform a task immediately (that is, as soon as you press **ENTER**), while statements are executed when a program is run. In TI Extended BASIC many commands can be used as statements, and most statements can be used as commands. A list of all the commands, statements, and functions is given in *Appendix B*, indicating the commands that can be used as statements and the statements that can be used as commands.

NEW

To remove a program from TI Extended BASIC to prepare the computer to accept a new program, use the NEW command. Programs are also removed from memory by the OLD command and the RUN command when used with a file name.

NUMBER and RESEQUENCE

When you are entering a program, the computer assigns line numbers for you if you enter the NUMBER command. If you wish to resequence the line numbers of a program after it is written, use the RESEQUENCE command.

LIST

To review the program that you have entered, use the LIST command. The program can be listed on the screen or to a peripheral device.

RUN

The RUN command instructs the computer to perform, or "execute," a program. The RUN command may be followed by a line number to have it start program execution at a specific line, or by a *device* and *filename* to load and execute a program from a diskette.

TRACE, UNTRACE, BREAK, UNBREAK, and CONTINUE

All of these commands are related to "debugging" a program, which is finding a problem that causes an error condition or an incorrect result. These commands are discussed further in the "Debugging and Error Handling" section of this chapter.

SAVE, OLD, MERGE, and DELETE

When you are finished working on a program, you may want to store it on a cassette or diskette for later use. The SAVE command, followed by the name of the storage device and a program name, performs this task for you. Then, when you wish to reuse, list, edit, or change a program, you can load it into memory with the OLD command. If a program has been saved using the merge option, you can combine it with a program already in memory with the MERGE command. When you have no further use for a program that has been saved on diskette, you can remove it with the DELETE command.

SIZE

The SIZE command lets you determine how much memory space is left, so you can decide whether to continue to add program lines or end the program and have a second program run from the first program with RUN used as a statement.

BYE

When you have finished using TI Extended BASIC, use the BYE command to return to the master title screen.

Several of the commands (RUN, BREAK, UNBREAK, TRACE, UNTRACE, and DELETE) can also be used as statements in programs.

ASSIGNMENTS AND INPUT

This section discusses statements in TI Extended BASIC that assign values to variables and enter data into programs.

LET and READ

If you know what values are to be assigned to variables, use LET or READ statements. LET is used when you are assigning a fairly small number of values or are calculating values to be assigned, and READ is used, in conjunction with DATA and RESTORE, when you are assigning numerous values.

INPUT and LINPUT

When you want the user of the program to assign values, it is customary to give a prompt that asks for the necessary information. INPUT allows you to give a prompt and accept input. INPUT only allows the entry of values at the bottom of the screen and cannot check to see that the data entered is the type of information the program expects. The final limitation on INPUT is that commas and quotation marks affect what is entered. With LINPUT, there is no editing of what is input, so commas and quotation marks can be input. Both INPUT and LINPUT may be used to input data from files on cassettes and diskettes.

ACCEPT

ACCEPT allows input from most screen positions. Using ACCEPT eliminates the necessity of entering data at the bottom of the screen and the "scrolling" of the INPUT statement. However, ACCEPT doesn't allow a prompt as the INPUT statement does. Therefore, a PRINT or DISPLAY statement must be included in the program to tell the user the type of entry that is required. ACCEPT can check the input to see that it is numeric, alphabetical, or specific characters. ACCEPT is for screen and keyboard use only.

CALL KEY and CALL JOYST

If pressing a single key is all that the program user is required to do, then CALL KEY can be used. For example, if a Y for "yes" or N for "no" is the required response, use the CALL KEY statement to accept the entry. CALL KEY does not display a character on the screen. It scans the keyboard or a portion of the keyboard to see if a key has been pressed. The major limitation of CALL KEY is that only a single keystroke is accepted. The data is not recorded as a character, but rather as the ASCII code for the character or as some other code. (See *Appendices C* and *J* for a list of the codes used.) If you wish to show the key that was pressed, you must use DISPLAY, PRINT, CALL VCHAR, or CALL HCHAR. The input from a Wired Remote Controller can be used with CALL JOYST. As with CALL KEY, the data is not displayed, and no scrolling takes place.

CALL CHARPAT, CALL COINC, CALL DISTANCE, CALL ERR, FOR-TO-STEP, CALL GCHAR, CALL POSITION, NEXT, CALL SPGET, and CALL VERSION

Each of these statements assigns one or more values to a variable. CALL CHARPAT assigns a value that specifies the pattern of a character. CALL COINC assigns a value to tell if sprites or a sprite and a point on the screen are at or near the same location on the screen. CALL DISTANCE indicates the distance between two sprites or a sprite and a point on the screen. CALL ERR specifies the error that occurred and where it occurred. CALL GCHAR reads what character is at a given screen location. CALL POSITION reads where a sprite is on the screen. CALL SPGET assigns the coded value of a speech phrase to a variable to be used with CALL SAY. CALL VERSION indicates the version of BASIC in use.

FOR-TO-STEP and NEXT deserve special comment. The FOR-TO-STEP statement sets the value of a variable so that it can be used to control the number of times a loop is executed. Each time NEXT is encountered, the value of the variable is changed. After the loop has been completed, the variable has a value that is the first value outside the range specified in the FOR-TO-STEP statement.

OUTPUT

This section discusses the TI Extended BASIC statements which are used to output data during program execution. Usually, output consists of displaying information on the screen, printing data on a printer, or saving data on an external device. However, output can also involve changing the color of the screen, changing the colors of characters, making noises, speaking, or sending data to peripheral devices.

PRINT, DISPLAY, PRINT...USING, DISPLAY...USING, and IMAGE

The two most frequently used output statements are PRINT and DISPLAY. The print separators (comma, semicolon, and colon) and the TAB function are used to control the placement of information as it is output. PRINT displays items at the bottom of the screen and scrolls them upward, one line at a time. With DISPLAY, you can display data almost anywhere on the screen without scrolling. DISPLAY can also clear the screen, erase characters on a line, and cause a beep.

PRINT...USING and DISPLAY...USING are like PRINT and DISPLAY except that the format of the printed or displayed characters is determined by the USING clause, possibly in conjunction with an IMAGE statement. The USING clause allows exact control of the format. PRINT and PRINT...USING, possibly in conjunction with IMAGE, are the only output statements that can be used to send data to an external device.

CALL HCHAR, CALL VCHAR, and CALL SPRITE

CALL HCHAR and CALL VCHAR place a character at any screen position and optionally repeat it horizontally or vertically. CALL SPRITE displays "sprites" on the screen. Sprites are graphics that can be moved smoothly in any direction and changed in pattern, size, and color. CALL SPRITE and the other statements related to sprites are discussed later in this chapter.

CALL SCREEN and CALL COLOR

In addition to displaying characters and data on the screen, you can change the color of the screen and the colors of the characters. CALL SCREEN sets the screen color. CALL COLOR specifies the foreground and background colors of characters or the color of sprites.

CALL SOUND and CALL SAY

CALL SOUND outputs sounds. A wide range of sounds is available. In addition, CALL SAY (possibly used with CALL SPGET) makes the computer speak if you have a *Solid State Speech*™ Synthesizer attached to your computer.

FUNCTIONS, SUBROUTINES, AND SUBPROGRAMS

TI Extended BASIC provides extensive functions and subprograms for handling numbers and characters. In addition, you may construct your own functions and write your own subprograms and subroutines.

Functions are TI Extended BASIC language elements that return a value, usually based on parameters given to the function. Many functions are mathematical in nature; others control or affect the result or output produced by the statements in which they occur. The TI Extended BASIC functions are ABS, ASC, ATN, CHR$, COS, EOF, EXP, INT, LEN, LOG, MAX, MIN, PI, POS, REC, RND, RPT$, SEG$, SGN, SIN, SQR, STR$, TAB, TAN, and VAL.

You can also define your own functions using DEF. Functions are used within TI Extended BASIC statements.

Built-in Functions

The following briefly discusses each built-in function.

Function	*Value Returned and Comments*
ABS	Absolute value of a numeric expression.
ASC	The numeric ASCII code of the first character of a string expression.
ATN	Trigonometric arctangent of a numeric expression given in radians.
CHR$	Character that corresponds to an ASCII code.
COS	Trigonometric cosine of a numeric expression given in radians.
EOF	End-of-file condition of a file.
EXP	Exponential value (e^x) of a numeric expression.
INT	Integer value of a numeric expression.
LEN	Number of characters in a string expression.
LOG	Natural logarithm of a numeric expression.
MAX	Larger of two numeric expressions.
MIN	Smaller of two numeric expressions.
PI	π with a value of 3.141592654.
POS	Position of the first occurrence of one string expression within another.
REC	Current record position in a file.
RND	Random number from 0 to 1.
RPT$	String expression equal to a number of copies of a string expression concatenated together.
SEG$	Substring of a string expression, starting at a specified point in that string and ending after a certain number of characters.
SGN	Sign of a numeric expression.
SIN	Trigonometric sine of a numeric expression given in radians.
SQR	Square root of a numeric expression.
STR$	String equivalent of a numeric expression.
TAB	Position for the next item in the *print-list* of PRINT, PRINT...USING, DISPLAY, or DISPLAY...USING.
TAN	Trigonometric tangent of a numeric expression given in radians.
VAL	Numeric value of a string expression which represents a number.

User-Defined Functions

DEF is used to define your own functions. Functions up to one line in length may be defined, with up to one argument. Longer functions may be constructed by having new definitions refer to previously defined functions. However, long functions might be more efficiently handled with subroutines or subprograms.

Subroutines

GOSUB and ON...GOSUB are used to call subroutines. A subroutine is a series of statements designed to perform a task and is normally used in a program when it performs a task several times. By using GOSUB or ON...GOSUB, you do not have to type the same lines of code several times. The subroutine can use the values of any variable in the program and change those values.

Built-in Subprograms

Built-in subprograms are TI Extended BASIC elements that perform special functions. They always are accessed with the CALL statement. The built-in subprograms are CHAR, CHARPAT, CHARSET, CLEAR, COINC, COLOR, DELSPRITE, DISTANCE, ERR, GCHAR, HCHAR, INIT, JOYST, KEY, LINK, LOAD, LOCATE, MAGNIFY, MOTION, PATTERN, PEEK, POSITION, SAY, SCREEN, SOUND, SPGET, SPRITE, VCHAR, and VERSION

Built-in subprograms perform many different tasks. Some of the subprograms affect the display and determine what key has been pressed on the keyboard.

Built-in Subprogram	*Action and Comments*
CLEAR	Clears the screen.
COLOR	Specifies the colors of characters in character sets or the color of sprites.
GCHAR	Returns the ASCII code of the character at a screen position.
HCHAR	Displays a character on the screen and optionally repeats it horizontally.
JOYST	Returns values indicating the position of the Wired Remote Controllers (optional).
KEY	Returns a code indicating the key that has been pressed.
SCREEN	Specifies the color of the screen.
VCHAR	Displays a character on the screen and optionally repeats it vertically.

Built-in subprograms can also define and control sprites.

Built-in Subprogram	*Action and Comments*
CHAR	Specifies the pattern for a character used for a sprite or a graphic.
COINC	Determines if two sprites or a sprite and a point on the screen are at or near the same location on the screen.
COLOR	Specifies the color of a sprite or a character set.
DELSPRITE	Deletes sprites.
DISTANCE	Determines the distance between two sprites or a sprite and a location.
LOCATE	Specifies the position of a sprite.
MAGNIFY	Changes the size of sprites.
MOTION	Specifies the motion of a sprite.
PATTERN	Specifies the character that defines a sprite.
POSITION	Determines the position of a sprite.
SPRITE	Defines sprites, specifying the character that defines them, their color, their position, and their motion.

A third category of built-in TI Extended BASIC subprograms involves sound and speech.

Built-in Subprogram	*Action and Comments*
SAY	Causes the computer to speak words when used in conjunction with the *Solid State Speech*™ Synthesizer.
SOUND	Generates sounds.
SPGET	Retrieves the codes that make speech.

Four built-in subprograms are only used with machine language subprograms obtained from Texas Instruments or another source written in TMS9900 machine language on another computer. Machine language subprograms cannot be written on the TI-99/4 Home Computer. Detailed instructions on the use of INIT, LINK, LOAD, and PEEK are provided with machine language subprograms.

Finally there are some miscellaneous built-in subprograms.

Built-in Subprogram	*Action and Comments*
CHARPAT	Returns a value that identifies the pattern of a character.
CHARSET	Resets characters 32 through 95 to their original pre-defined patterns and colors.
ERR	Returns values which give information about an error that has occurred.
VERSION	Specifies the version of BASIC that is being used.

User-Written Subprograms

You may write your own subprograms. They are a series of statements designed to perform a task. They may be used in a program when you expect to perform the task several times or to perform the same task in several different programs. Using the MERGE option when you save a subprogram allows it to be included in other programs.

When a subprogram is in a program, it must follow the main program. The structure of a program must be as follows:

Start of Main Program . . . Subprogram Calls . . .	
End of Main Program	The program will stop here without a STOP or END statement.
Start of First Subprogram . . .	Subprograms are optional.
End of First Subprogram	Nothing may appear between subprograms except remarks and the END statement.
Start of Second Subprogram . . .	
End of Second Subprogram	Only remarks and END may appear after the subprograms.
End of Program	

Subprograms are called by the use of CALL followed by the subprogram's name and an optional list of parameters and values. The first line of a subprogram is SUB, followed by the name of the subprogram and optionally followed by a list of parameters.

The subprograms you write are not part of the main program. They cannot use the values of variables in the main program, so any values that are needed must be supplied by the parameter list in the CALL statement. Variable names may be duplicates of those in the main program or other subprograms without affecting the values of the variables in the main program or other subprograms. Subprograms may call other subprograms, but must not call themselves, either directly or indirectly.

SUBEND must be the last statement in a subprogram. When that statement is executed, control returns to the statement following the statement that called the subprogram. Control may also be returned by the SUBEXIT statement.

SOUND, SPEECH, AND COLOR

You may highlight important sections of your programs's output through the use of sounds, speech, and colors. This "human engineering" makes the program easier and more interesting to use.

CALL SOUND

SOUND outputs sounds. Tones may be output in lengths of from .001 to 4.25 seconds at volumes from 0 (loudest) to 30 (softest). The frequency range is from 110 (A below low C) to 44,733 (above the range of human hearing). In addition, 8 noises are available. Up to three tones and one noise may be produced at the same time. *Appendix D* lists the frequencies that are used to produce the musical notes.

CALL SAY and CALL SPGET

SAY produces speech when a Texas Instruments *Solid State Speech*™ Synthesizer (sold separately) is attached to the console. You can choose among 373 letters, numbers, words, and phrases (listed in *Appendix L*). In addition, you can construct new words from old by combining words. For example, SOME + THING produces "something" and THERE + FOUR produces "therefore."

SPGET is used to retrieve the speech codes that produce speech. These patterns can then be used to produce more natural speech and can be used to change words. Because making new words is a complex process, it is not discussed in this manual. However, suffixes can be added rather simply. *Appendix M* tells how to add the suffixes ING, S, and ED to any word, so that words such as ANSWERING, ANSWERS, ANSWERED, INSTRUCTING, INSTRUCTS, and INSTRUCTED are included in the computer's vocabulary.

CALL COLOR and CALL SCREEN

COLOR changes the colors of character sets and determines sprite colors. SCREEN specifies the color of the screen as one of the sixteen colors available on the TI-99/4 Home Computer.

SPRITES

Sprites are graphics that can be displayed and moved on the screen. One advantage that sprites have over other characters is that they can be at any of 49,152 positions of 192 rows and 256 columns rather than one of the 768 positions of 24 rows and 32 columns used by statements such as CALL VCHAR and CALL HCHAR. Because of this greater resolution, sprites can move more smoothly than characters. Also, once set in motion, sprites can continue to move without further program control.

CALL SPRITE

CALL SPRITE defines sprites. This subprogram specifies the character pattern that sprites use, their color, their position, and, optionally, their motion.

CALL CHAR and CALL MAGNIFY

Although you may use any of the predefined characters, numbers 32 through 95, as a sprite, CALL CHAR is generally used to define a new pattern for a sprite. Up to four 8 by 8 dot characters may be used to form a sprite. The MAGNIFY subprogram controls the resolution and size of sprites.

CALL COLOR, CALL LOCATE, CALL PATTERN, and CALL MOTION

Once a sprite is set up, it can be altered by various subprograms. COLOR changes the color of a sprite. LOCATE moves the sprite to a new position. PATTERN changes the character that defines a sprite. MOTION alters the motion of a sprite.

CALL COINC, CALL DISTANCE, and CALL POSITION

Three subprograms provide information about sprites while a program is running. COINC returns a value that indicates if sprites or a sprite and a point on the screen are at or near the same place on the screen. DISTANCE returns a value that specifies the distance between two sprites or a sprite and a point on the screen. POSITION returns values that indicate the position of a sprite.

CALL DELSPRITE

CALL DELSPRITE allows you to delete sprites. If you prefer, you may "hide" sprites by locating them off the bottom of the screen.

DEBUGGING

Debugging a program is finding logical or typing errors in a program. BREAK, CONTINUE, TRACE, ON BREAK, UNBREAK, UNTRACE, and **SHIFT C** (CLEAR) are most often used in debugging.

BREAK, ON BREAK, CONTINUE, and UNBREAK

BREAK causes the computer to stop program execution so that you can print the values of variables or change their values. BREAK also resets characters to their standard colors (black on transparent), restores the standard screen color (cyan), restores the standard characters (32-95) to their standard representation, and deletes sprites.

ON BREAK tells the computer what to do if a break occurs. You can use this statement to tell the computer to ignore breakpoints that you have entered in the program. CONTINUE causes the computer to continue program execution after a breakpoint. UNBREAK cancels any breakpoints set with BREAK. *Note*: If you have put ON BREAK CONTINUE, the computer will not stop when you press **SHIFT C** (CLEAR).

TRACE and UNTRACE

TRACE causes the computer to display each line number before the statement(s) on that line is (are) executed. Using this statement allows you to follow the sequence of operation of a program. UNTRACE cancels the operation of TRACE.

ERROR HANDLING

You may include statements in a program to handle errors that occur while the program is running.

CALL ERR, ON ERROR, ON WARNING, and RETURN

CALL ERR returns information indicating where an error has occured and what the error is. *Appendix N* lists the error codes that are returned. ON ERROR specifies what the computer does if an error occurs. ON WARNING specifies what the computer does if a condition arises that would normally cause a warning message to be issued. RETURN is used with ON ERROR in addition to its use with GOSUB. It repeats execution of the statement that caused the error, returns to the statement following the one that caused the error, or transfers control to some other part of the program that avoids the error that has occurred.

PROGRAM ENTRY EXAMPLE

Now that you've had a brief overview of the features of TI Extended BASIC, you may enjoy reviewing or even entering and experimenting with a demonstration program. This section demonstrates a number of the useful features of TI Extended BASIC. By following the suggestions in this section, you can learn some useful shortcuts in the entry process.

This program allows you to play a game called Codebreaker. In playing it, you determine the length of a code (1 to 8 digits). Then you decide the range of digits that may be included in the code (up to ten). The computer selects the digits in the code without repeating digits. You then guess what the digits are and their sequence. After each guess, the computer tells you how many digits you guessed correctly and how many are in the correct place. (If you repeat a digit in your guess, it is counted as right each time it appears.) Using this information, you guess again. You win when you guess all the digits correctly and place them in the proper sequence.

For example, suppose you've chosen to play the game using four digits with each digit being any one of nine numbers (0, 1, 2, 3, 4, 5, 6, 7, or 8). The code the computer chooses might be 0743, which you are trying to break. Here is a possible sequence of guesses.

GUESS	*RIGHT*	*PLACE*	*EXPLANATION OF THE COMPUTER'S RESPONSE*
0000	4	1	0 is right four times, once in the right place.
1234	2	0	3 and 4 are right, but not in the right place.
5678	1	0	7 is right, but not in the right place.
2348	2	1	3 and 4 are right, and 4 is in the right place.
0347	4	2	All right, 0 and 4 in the right place.
3047	4	1	All right, 4 in the right place
0734	4	2	All right, 0 and 7 in the right place.
0743	4	4	All right, all in the right place. You win.

To begin entering the example, turn on any peripheral devices you have connected to the computer. Insert the TI Extended BASIC Command Module and turn on the computer. Press any key to go to the master selection list. **Press 3** to select TI Extended BASIC.

In the following, the characters you type and the keys you press are <u>UNDERLINED</u>.

CODEBREAKER Program Entry

COMMENTS	DISPLAY	
	* READY *	
Automatically numbers the program lines.	>NUM	ENTER
Title and language.	>100 REM CODEBREAKER XBASIC	ENTER
Reserves room for the codes and guesses.	>110 DIM CODE$(8),GUESS$(8)	ENTER
Makes the codes random.	>120 RANDOMIZE	ENTER
Clears the screen, beeps, and puts the title CODEBREAKER on the 11th row starting in the 9th column.	>130 DISPLAY AT(11,9)BEEP ERA SE ALL:"CODEBREAKER"	ENTER
REDO repeats whatever was done before **ENTER** was last pressed. Using the edit keys [**SHIFT G** (INSERT), **SHIFT F** (DELETE), and the arrows], change line 130 to: 140 DISPLAY AT(19,1)BEEP:" NUMBER OF CODES? (1-8)".	>140	SHIFT R
Beeps and displays NUMBER OF CODES? (1-8) on the 19th row starting at the first column.	140 DISPLAY AT(19,1)BEEP:"NU MBER OF CODES? (1-8)"	ENTER
Press **SHIFT R** (REDO) again. Now change line 140 to: 150 DISPLAY AT(21,6)BEEP:"DIGITS EACH CODE?".	>	SHIFT R
Beeps and displays DIGITS EACH CODE? on the 21st row starting at the 6th column.	150 DISPLAY AT(21,6)BEEP:"DI GITS EACH CODE?"	ENTER
Accepts into CODES an entry on the 19th line, 24th column, allowing only digits to be entered.	>160 ACCEPT AT(19,24)VALIDATE (DIGIT):CODES	ENTER
Change line 160 to: 170 ACCEPT AT(21,24) VALIDATE(DIGIT): DIGITS.	>	SHIFT R
Accepts into DIGITS an entry on the 21st line, 24th column, allowing only digits to be entered.	170 ACCEPT AT(21,24)VALIDATE (DIGIT):DIGITS	ENTER

COMMENTS	DISPLAY
Displays the program as it is currently entered.	>LIST

```
100 REM CODEBREAKER XBASIC
110 DIM CODE$(8),GUESS$(8)
120 RANDOMIZE
130 DISPLAY AT(11,9)BEEP ERA
SE ALL:"CODEBREAKER"
140 DISPLAY AT(19,1)BEEP:"NU
MBER OF CODES? (1-8)"
150 DISPLAY AT(21,6)BEEP:"DI
GITS EACH CODE?"
160 ACCEPT AT(19,24)VALIDATE
(DIGIT):CODES
170 ACCEPT AT(21,24)VALIDATE
(DIGIT):DIGITS
```

Runs the program. >RUN

Screen clears, then this appears:

```
        CODEBREAKER

NUMBER OF CODES? (1-8) ■

      DIGITS EACH CODE?
```

Enter anything except a digit. The computer beeps and does not accept it. Enter 4. The cursor moves down to the second prompt. Enter 10. The program ends and you can continue entry.

COMMENTS	DISPLAY	
	* READY *	
Numbers lines starting with 180.	>NUM 180	
Checks to see that there will be enough digits for the number of codes. If CODES is less than or equal to DIGITS, control passes to the next line. If CODES is greater than DIGITS, the message NO MORE CODES THAN DIGITS is displayed on the last line of the screen, and control is transferred to line 160 again.	>180 IF CODES>DIGITS THEN DIS PLAY AT(24,2)BEEP:"NO MORE C ODES THAN DIGITS"::GOTO 160	ENTER

Starts the loop to choose the codes. The words after the exclamation point are a comment.	>190 FOR A=1 TO CODES !CHOOSE CODES	ENTER
Chooses codes at random.	>200 CODE$(A)=STR$(INT(RND*DIGITS))	ENTER
Starts the loop to prevent duplicate codes.	>210 FOR B=0 TO A-1 !CHECK FOR DUPLICATES	ENTER
Checks for duplicates. Chooses a new code if there is a duplicate.	>220 IF CODE$(A)=CODE$(B) THEN 200	ENTER
Finishes duplicate check loop.	>230 NEXT B	ENTER
Finishes code choice loop.	>240 NEXT A	ENTER
Sets a variable to keep track of where information is displayed on the screen.	>250 ROW=2	ENTER
Clears the screen and displays a column heading on the top line.	>260 DISPLAY AT(1,1)ERASE ALL :"GUESS RIGHT PLACE"	ENTER
REDO line 260 so it reads: 270 DISPLAY AT(24,3):"ENTER 'X' FOR SOLUTION".	>270	SHIFT R
Displays an instruction at the bottom of the screen.	270 DISPLAY AT(24,3):"ENTER 'X' FOR SOLUTION"	ENTER
Numbers lines starting at 280.	>NUM 280	ENTER
Accepts the guess at the proper row.	>280 ACCEPT AT(ROW,1):C$	ENTER
Checks for giving up or resetting.	>290 IF C$="X" THEN 470 !GIVE UP OR RESET	ENTER
Begins loop to break up the guess to check it for accuracy.	>300 FOR D=1 TO CODES !BREAK UP GUESS	ENTER
Separates guess into individual digits.	>310 GUESS$(D)=SEG$(C$,D,1)	ENTER
Completes loop to separate guess.	>320 NEXT D	ENTER
Sets RIGHT and PLACE to zero.	>330 RIGHT,PLACE=0	ENTER
Begins outside loop to check the guess against the code.	>340 FOR E=1 TO CODES !CHECK GUESS FOR CORRECTNESS	ENTER
Begins inside loop to check guess.	>350 FOR F=1 TO CODES	ENTER
If a guess doesn't match a code, goes to the next line. If a guess matches a code, adds one to the number correct. Then if the guess is in the correct place, adds one to the number in the correct place.	>360 IF CODE$(E)=GUESS$(F) THEN RIGHT=RIGHT+1::IF E=F THEN PLACE=PLACE+1	ENTER

Completes inside loop.	>370 NEXT F	ENTER
Completes outside loop.	>380 NEXT E	ENTER
Displays the number of digits that are correct.	>390 DISPLAY AT(ROW,14):RIGHT	ENTER
REDO line 390 to be: 400 DISPLAY AT (ROW,22):PLACE.	>400	SHIFT R
Displays the number of digits that are in the correct place.	400 DISPLAY AT(ROW,22):PLACE	ENTER
Numbers lines starting at 410.	>NUM 410	ENTER
Checks to see if the code has been solved. If it has, goes to the next line. If it has not, adds one to the row. Then if the row is more than 22, goes to line 470 and gives the solution. Otherwise, returns to line 280 to accept another guess.	>410 IF PLACE<>CODES THEN ROW=ROW+1::IF ROW>22 THEN 470 ELSE 280	ENTER
Displays the win message with the number of guesses at the 23rd row starting at the first column.	>420 DISPLAY AT(23,1)BEEP:"YOU WIN WITH";ROW-1;"GUESSES."	ENTER
REDO line 420 to be: 430 DISPLAY AT(24,1) BEEP:"PLAY AGAIN? (Y/N) Y".	>430	SHIFT R
Displays the prompt PLAY AGAIN? (Y/N) Y at the 24th row starting at the first column.	430 DISPLAY AT(24,1)BEEP:"PLAY AGAIN? (Y/N) Y"	ENTER
Numbers lines starting at 440.	>NUM 440	ENTER
Accepts an entry into X$ on the 24th row, 19th column. Does not remove any character that is already there (in this case, a Y from the DISPLAY statement in line 430), accepts only one character, beeps, and accepts only Y or N. Pressing **ENTER** at this point when the program is running confirms the Y that was displayed by line 430.	>440 ACCEPT AT(24,19)SIZE(-1)BEEP VALIDATE("YN"):X$	ENTER
If Y is entered, returns to line 190 and chooses a new code for another game.	>450 IF X$="Y" THEN 190	ENTER
Stops the program.	>460 STOP	ENTER

Displays the message THE CODE IS at the 23rd row, 1st column.	>470 DISPLAY AT(23,1)BEEP:"THE CODE IS" !LOSE, GIVE UP, OR RESET	ENTER
Begins a loop to display the digits.	>480 FOR G=1 TO CODES	ENTER
Displays the digits.	>490 DISPLAY AT(23,12+G):CODE$(G)	ENTER
Finishes the loop.	>500 NEXT G	ENTER
Leave the number mode.	>510	ENTER
Press DOWN ARROW as if to edit line 430 so you can use **SHIFT R** (REDO).	>430 430 DISPLAY AT(24,1)BEEP:"PLAY AGAIN? (Y/N) Y"	DOWN ARROW ENTER
Press REDO. Line 510 is a duplicate of line 430, so change the line number to 510.	>	SHIFT R
Displays the prompt PLAY AGAIN? (Y/N) Y at the 24th row starting at the 1st column.	510 DISPLAY AT(24,1)BEEP:"PLAY AGAIN? (Y/N) Y"	ENTER
Press DOWN ARROW as if to edit line 440 so you can use **SHIFT R** (REDO).	>440 440 ACCEPT AT(24,19)SIZE(-1) BEEP VALIDATE("YN"):X$	DOWN ARROW ENTER
Press REDO. Line 520 is a duplicate of line 440, so change the line number to 520.	>	SHIFT R
Accepts an entry into X$ on the 24th row, 19th column. Does not remove any character that is already displayed (in this case a Y from the DISPLAY statement in line 510), accepts only one character, beeps, and accepts only Y or N. Pressing **ENTER** at this point when the program is running confirms the Y that was displayed by line 510.	>520 ACCEPT AT(24,19)SIZE(-1) BEEP VALIDATE("YN"):X$	ENTER
If Y is entered, returns to line 130, allows changing the number of digits in a code and the number of acceptable digits, and starts a new game.	>530 IF X$="Y" THEN 130	ENTER

Before running a program, you should proofread it. Here is a list of the entire program for you to check against your program list.

```
100 REM CODEBREAKER XBASIC
110 DIM CODE$(8),GUESS$(8)
120 RANDOMIZE
130 DISPLAY AT(11,9)BEEP ERA
SE ALL:"CODEBREAKER"
140 DISPLAY AT(19,1)BEEP:"NU
MBER OF CODES? (1-8)"
150 DISPLAY AT(21,6)BEEP:"DI
GITS EACH CODE?"
160 ACCEPT AT(19,24)VALIDATE
(DIGIT):CODES
170 ACCEPT AT(21,24)VALIDATE
(DIGIT):DIGITS
180 IF CODES>DIGITS THEN DIS
PLAY AT(24,2)BEEP:"NO MORE C
ODES THAN DIGITS"::GOTO 150
190 FOR A=1 TO CODES !CHOOSE
 CODES
200 CODE$(A)=STR$(INT(RND*DI
GITS))
210 FOR B=0 TO A-1 !NO DUPLI
CATES
220 IF CODE$(A)=CODE$(B) THE
N 200
230 NEXT B
240 NEXT A
250 ROW=2
260 DISPLAY AT(1,1)ERASE ALL
:"GUESS       RIGHT   PLACE"
270 DISPLAY AT(24,3):"ENTER
'X' FOR SOLUTION"
280 ACCEPT AT(ROW,1):C$
290 IF C$="X" THEN 470 !GIVE
 UP OR RESET
300 FOR D=1 TO CODES !BREAK
UP GUESS
```

```
310 GUESS$(D)=SEG$(C$,D,1)
320 NEXT D
330 RIGHT,PLACE=0
340 FOR E=1 TO CODES !CHECK
GUESS
350 FOR F=1 TO CODES
360 IF CODE$(E)=GUESS$(F) TH
EN RIGHT=RIGHT+1::IF E=F THE
N PLACE=PLACE+1
370 NEXT F
380 NEXT E
390 DISPLAY AT(ROW,14):RIGHT
400 DISPLAY AT(ROW,22):PLACE
410 IF PLACE<>CODES THEN ROW
=ROW+1::IF ROW>22 THEN 470 E
LSE 280
420 DISPLAY AT(23,1)BEEP:"YO
U WIN WITH";ROW-1;"GUESSES."
430 DISPLAY AT(24,1)BEEP:"PL
AY AGAIN? (Y/N) Y"
440 ACCEPT AT(24,19)SIZE(-1)
BEEP VALIDATE("YN"):X$
450 IF X$="Y" THEN 190
460 STOP
470 DISPLAY AT(23,1)BEEP:"TH
E CODE IS" !LOSE, GIVE UP, O
R RESET
480 FOR G=1 TO CODES
490 DISPLAY AT(23,12+G):CODE
$(G)
500 NEXT G
510 DISPLAY AT(24,1)BEEP:"PL
AY AGAIN? (Y/N) Y"
520 ACCEPT AT(24,19)SIZE(-1)
BEEP VALIDATE("YN"):X$
530 IF X$="Y" THEN 130
```


Now run the program by typing RUN and pressing **ENTER**. Choose 4 codes with 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) possible in each code. Guessing the code in six tries is excellent. Finding it in eight is very good.

If you wish to use the program again, save it on diskette or cassette. To save it on cassette, make sure the cassette player is connected. Then enter SAVE CS1 and follow the instructions that appear on the screen.

To save the program on diskette, enter SAVE DSK1.*filename* with whatever *filename* you wish to use to save it, such as CODEBREAK.

After saving the program, or if you do not wish to save the program, enter NEW. The program is removed and you may enter another program.

If you have saved the program, you can easily reload it into the computer's memory for reuse or further editing. Reload the program from a cassette by entering OLD CS1 and then following the instructions that appear on the screen. Reload the program from diskette by entering OLD DSK1.*filename* using whatever *filename* you used to save it.

When you have finished using TI Extended BASIC, enter BYE to return to the master title screen.

CHAPTER 3

TI Extended BASIC Conventions

This chapter discusses the format that TI Extended BASIC programs must take and the ways in which TI Extended BASIC functions.

TI EXTENDED BASIC CONVENTIONS

RUNNING A PROGRAM ON POWERUP

If a program named LOAD is on the diskette in disk drive 1 when TI Extended BASIC is chosen, that program is loaded and run. The effect is the same as if you had entered RUN "DSK1.LOAD". If the program does not exist, there is a momentary delay while TI Extended BASIC looks for it.

FILES

Files are groups of data put on external devices. The most common files are on cassettes or diskettes, but data sent through external devices such as the RS232 Interface and the optional thermal printer are also considered to be files by TI Extended BASIC.

LINE NUMBERS

Line numbers are required in TI Extended BASIC programs. Line numbers specify the order in which lines are executed and are used to identify what lines to execute next when using IF-THEN-ELSE, GOTO, GOSUB, ON ERROR, ON...GOTO, and ON...GOSUB. Line numbers may also be used by BREAK, LIST, NUM, RESTORE, RETURN, and RUN. Line numbers may be any integer from 1 through 32767.

The computer automatically generates line numbers if you issue the NUM command. When not followed by a line number, it provides line numbers starting at 100, incrementing each subsequent line by 10. You may resequence line numbers with the RES command.

LINES

Lines may be up to 140 characters long, including the line number and spaces. If you have reached the end of a line, additional characters you enter replace the 140th character. It is possible to make a line longer than 140 characters in the Edit Mode by the use of **SHIFT G** (INSERT).

SPECIAL SYMBOLS

Special symbols separate statements and remarks on the same line. A line of TI Extended BASIC consists of a line number, one or more TI Extended BASIC statements, and an optional remark. For example:

```
100 FOR A=1 TO 100::PRINT A;SQR(A)::NEXT A !PRINT SQUARE ROOTS
```

The statement separator symbol, a double colon (::), is used to separate statements on the same line. The tail remark symbol, an exclamation point (!), is used to separate an explanatory remark from the rest of the line. Remarks are not executed when the program is run.

SPACES

Spaces are required in TI Extended BASIC between the elements that make up statements to enable the computer to distinguish variable names from TI Extended BASIC elements. However, spaces are not required before or after relational symbols or before or after the tail remark symbol or the statement separator symbol. You may insert extra spaces when inputting commands and statements, but they are deleted by TI Extended BASIC. When listing programs, TI Extended BASIC may add spaces around the tail remark symbol and statement separator symbol.

NUMERIC CONSTANTS

Numeric constants may be entered with any number of digits. However, they are rounded to 13 or 14 digits by the computer due to the internal storage method used by the computer, and are normally displayed as a maximum of 10 digits. For extremely large or small numbers, it is usually more convenient to use scientific notation to enter numbers. The computer normally uses scientific notation when printing very large or small numbers.

In scientific notation, a number is given as a mantissa (a number with one place to the left of the decimal point) times 10 raised to an integer power. 15 is expressed in scientific notation as 1.5×10^1. 150 is expressed as 1.5×10^2; $-1{,}500$ is expressed as -1.5×10^3; 156,789,000,000,000 is expressed as 1.56789×10^{14}; and 0.156789 is expressed as 1.56789×10^{-1}. In TI Extended BASIC, The "x 10" is represented by "E". Thus 1.5×10^3 becomes 1.5E3.

Numeric constants are defined in the range -9.9999999999999E127 to -1E-128, 0, and 1E-128 to 9.9999999999999E127. If the exponent of a calculated number is greater than 99, then ** is normally printed or displayed as the power. The entire exponent is kept internally and can be displayed with a USING clause in a PRINT or DISPLAY statement.

STRING CONSTANTS

String constants in TI Extended BASIC can be up to one input line long. If the string is enclosed in quotation marks, quotation marks in the string are represented by double quotation marks.

VARIABLES

Variables in TI Extended BASIC may consist of one to 15 characters. The first character of a variable must be a letter of the alphabet, the at symbol (@), or an underline (_). Subsequent characters may be these symbols plus any of the digits. The last character of a string variable must always be a dollar sign ($). Variables are either scalar or arrays with up to seven dimensions.

TI EXTENDED BASIC CONVENTIONS

Certain words are reserved for use by TI Extended BASIC. They are the commands, statements, functions, and operators that make up the language. These words may not be used as a variable name, but they may make up part of a variable name. The following is a complete list of the words reserved for TI Extended BASIC.

ABS	EOF	NUMBER	SEQUENTIAL
ACCEPT	ERASE	NUMERIC	SGN
ALL	ERROR	OLD	SIN
AND	EXP	ON	SIZE
APPEND	FIXED	OPEN	SQR
ASC	FOR	OPTION	STEP
AT	GO	OR	STOP
ATN	GOSUB	OUTPUT	STR$
BASE	GOTO	PERMANENT	SUB
BEEP	IF	PI	SUBEND
BREAK	IMAGE	POS	SUBEXIT
BYE	INPUT	PRINT	TAB
CALL	INT	RANDOMIZE	TAN
CHR$	INTERNAL	READ	THEN
CLOSE	LEN	REC	TO
CON	LET	RELATIVE	TRACE
CONTINUE	LINPUT	REM	UALPHA
COS	LIST	RES	UNBREAK
DATA	LOG	RESEQUENCE	UNTRACE
DEF	MAX	RESTORE	UPDATE
DELETE	MERGE	RETURN	USING
DIGIT	MIN	RND	VAL
DIM	NEW	RPT$	VALIDATE
DISPLAY	NEXT	RUN	VARIABLE
ELSE	NOT	SAVE	WARNING
END	NUM	SEG$	XOR

The following are examples of valid variable names:

Numeric: X, A9, ALPHA, BASE_PAY V(3), T(X,Y,Z,Q,A,R,P6), TABLE(Q37,M/4)

String: S$, YZ2$, NAME$, Q5$(X,7,L/2), ADDRESS$(4)

NUMERIC EXPRESSIONS

Numeric expressions are constructed from numeric constants, numeric variables, and functions using the arithmetic operators for addition (+), subtraction (−), multiplication (*), division (/), and exponentiation (∧).

The minus sign (−) can be used either to indicate subtraction or as a unary minus. As a unary minus, it reverses the sign of what follows it. For example, −3∧2 is equal to −9 as it is taken to mean −(3∧2).

The normal hierarchy for evaluating a numeric expression is exponentiation, followed by multiplication and division, and then by addition and subtraction. However, any part of a numeric expression that is enclosed in parentheses is evaluated first. The following shows the effect of parentheses on determining the value of an expression:

Expression	*Intermediate Results*		*Final Value*
4+2∧2/2−6	4+4/2−6	4+2−6	0
(4+2)∧2/2−6	6∧2/2−6	36/2−6	12
4+2∧2/(2−6)	4+4/(−4)	4−1	3

STRING EXPRESSIONS

String expressions are constructed from string variables, string constants, and function references using the operation for concatenation (&) to combine strings. If a constructed string exceeds a length of 255 characters, the extra characters on the right are truncated and a warning message is issued. The following is an example of concatenation:

```
100 A$="HI"&" THERE!"
```

A$="HI"&" THERE!" sets A$ equal to "HI THERE!".

RELATIONAL EXPRESSIONS

Relational expressions are most often used in the IF-THEN-ELSE statement, but may be used anywhere that numeric expressions are allowed. A relational expression has a value of −1 if it is true and a value of 0 if it is false. Relational operations are performed from left to right, after all arithmetic operations are completed and before string concatenation (the ampersand operator). The relational expressions are:

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

The following examples illustrate the use of relational expressions:

IF X<Y THEN 200 ELSE GOSUB 420 next executes line 200 if X is less than Y. If X is greater than or equal to Y, then the statement GOSUB 420 is executed.

```
>100 IF X<Y THEN 200 ELSE GO
 SUB 420
```

IF L(C) = 12 THEN C = S + 1 ELSE COUNT = COUNT + 1::GOTO 140 sets C equal to S plus 1 if L(C) equals 12. If L(C) is not equal to 12, then COUNT is set equal to COUNT plus 1 and line 140 is executed next.

```
>100 IF L(C)=12 THEN C=S+1 E
 LSE COUNT=COUNT+1::GOTO 140
```

A = 2<5 sets A equal to − 1 as it is true that 2 is less than 5.

```
>100 A=2<5
```

PRINT "THIS" = "THAT" prints 0 as it is not true that "THIS" is equal to "THAT".

```
>100 PRINT "THIS"="THAT"
```

A = B = 7 sets A equal to − 1 if B is equal to 7, and to 0 if B is not equal to 7. There is no effect on B. Note that this is not the same as the usual arithmetical meaning of A = B = 7.

```
>100 A=B=7
```

LOGICAL EXPRESSIONS

Logical expressions are used with relational expressions. The logical operators are AND, OR, NOT and XOR. If true, logical expressions are given a value of − 1. If false, they are given a value of 0. The order of precedence for logical expressions, from highest to lowest, is NOT, XOR, AND, and OR.

A logical expression using AND is true if both its left and right clauses are true.

A logical expression using OR is true if either its left clause is true, its right clause is true, or both its left and right clauses are true.

A logical expression using NOT is true if the clause following it is not true.

A logical expression using XOR (exclusive or) is true if either its left or its right clause is true, but not *both* its left and right clauses are true.

The following examples illustrate the use of logical expressions:

IF 3<4 AND 5<6 THEN L = 7 sets L equal to 7 since 3 is less than 4 and 5 is less than 6.

```
>100 IF 3<4 AND 5<6 THEN L=7
```

IF 3<4 AND 5>6 THEN L = 7 does not set L equal to 7 because 3 is less than 4, but 5 is not greater than 6.

```
>100 IF 3<4 AND 5>6 THEN L=7
```

IF 3<4 OR 5>6 THEN L = 7 sets L equal to 7 because 3 is less than 4.

```
>100 IF 3<4 OR 5>6 THEN L=7
```

IF 3<4 XOR 5>6 THEN L = 7 sets L equal to 7 because 3 is less than 4 and 5 is not greater than 6.

```
>100 IF 3<4 XOR 5>6 THEN L=7
```

IF 3<4 XOR 5<6 THEN L = 7 does not set L equal to 7 because 3 is less than 4 and 5 is less than 6.

```
>100 IF 3<4 XOR 5<6 THEN L=7
```

IF NOT 3 = 4 THEN L = 7 sets L equal to 7 because 3 is not equal to 4.

```
>100 IF NOT 3=4 THEN L=7
```

IF NOT 3 = 4 AND (NOT 6 = 5 XOR 2 = 2) THEN 200 does not pass control to line 200 because while it is true that 3 is not equal to 4, it is true that both 6 is not equal to 5 and 2 is equal to 2, so the clause in parentheses is not true.

```
>100 IF NOT 3=4 AND (NOT 6=5
 XOR 2=2) THEN 200
```

IF (A OR B) AND (C XOR D) THEN 200 passes control to line 200 if either A or B or both A and B are true (equal to − 1), and C or D, but not both C and D are true (equal to − 1).

```
>100 IF (A OR B) AND (C XOR
 D) THEN 200
```

The logical operators can also be used directly on numbers. They convert the numbers to binary notation, perform the designated operation on a bit level, and then convert the result back to decimal representation. A more detailed discussion of the use of logical operators with numbers can be found in a mathematics or engineering text dealing with logic.

The numbers must be from − 32,768 to 32,767, represented in binary notation as from 1000000000000000 to 0111111111111111, with negative numbers given in 2's complement form signified by a 1 in the most significant bit. In binary notation, each place is an additional power of 2 rather than an additional power of 10 as in decimal notation. The following shows numbers in both decimal and binary notation.

DECIMAL PLACE				BINARY PLACE															
-	100	10	1	-	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	0	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
	0	2	5	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
-	0	1	3	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1

The above is the equivalent to

$1_{10} = 0000000000000001_2 = 1_2$ $\quad 25_{10} = 0000000000011001_2 = 11001_2$

$6_{10} = 0000000000000110_2 = 110_2$ $\quad -13_{10} = 1111111111110011_2$

AND places a 1 in the corresponding binary position if there is a 1 in both the number preceeding and following it. Otherwise it places a zero.

OR places a 1 in the corresponding binary position if there is a 1 in either the number preceeding it or following it or both. Otherwise it places a zero.

XOR places a 1 in the corresponding binary position if there is a 1 in either the number preceeding it or following it but not both. Otherwise it places a zero.

NOT places a 1 in the corresponding binary position if there is a zero in the number following it. Otherwise it places a zero.

The following illustrate the result of the logical operators when used on numbers.

	DECIMAL	BINARY		DECIMAL	BINARY
A:	1	0000000000000001	A:	1	0000000000000001
B:	2	0000000000000010	B:	3	0000000000000011
A AND B:	0	0000000000000000	A AND B:	1	0000000000000001
A:	6	0000000000000110	A:	47	0000000000101111
B:	5	0000000000000101	B:	62	0000000000111110
A AND B:	4	0000000000000100	A AND B:	46	0000000000101110

	DECIMAL	BINARY		DECIMAL	BINARY
A:	1	0000000000000001	A:	1	0000000000000001
B:	2	0000000000000010	B.	3	0000000000000011
A OR B:	3	0000000000000011	A OR B:	3	0000000000000011
A:	6	0000000000000110	A:	47	0000000000101111
B:	5	0000000000000101	B:	62	0000000000111110
A OR B:	7	0000000000000111	A OR B:	63	0000000000111111

	DECIMAL	BINARY		DECIMAL	BINARY
A:	1	0000000000000001	A:	1	0000000000000001
B:	2	0000000000000010	B:	3	0000000000000011
A XOR B:	3	0000000000000011	A XOR B:	2	0000000000000010
A:	6	0000000000000110	A:	47	0000000000101111
B:	5	0000000000000101	B:	62	0000000000111110
A XOR B:	3	0000000000000011	A XOR B:	17	0000000000010001

	DECIMAL	BINARY		DECIMAL	BINARY
A:	1	0000000000000001	A:	2	0000000000000010
NOT A:	-2	1111111111111110	NOT A:	-3	1111111111111101
A:	6	0000000000000110	A:	47	0000000000101111
NOT A:	-7	1111111111111001	NOT A:	-48	1111111111010000

CHAPTER 4

Reference Section

This chapter is an alphabetical list of all of the TI Extended BASIC commands, statements, and functions, with a detailed explanation of how each works. Examples and sample programs are included wherever necessary for clarity.

In the format of the elements, key words are CAPITALIZED. Variables are in *italics*. Optional portions are enclosed in [brackets]. Items that may be repeated are indicated by elipses (...). Alternative forms are presented one above the other.

Appendix A contains a list of the illustrative programs. The Index gives the pages on which each TI Extended BASIC element is used in an illustrative program.

ABS

Format

ABS(*numeric-expression*)

Description

The ABS function gives the absolute value of *numeric-expression*. If *numeric-expression* is positive, ABS gives the value of numeric expression. If *numeric-expression* is negative, ABS gives its negative (a positive number). If *numeric-expression* is zero, ABS returns zero. The result of ABS is always a non-negative number.

Examples

PRINT ABS(42.3) prints 42.3.

```
>100 PRINT ABS(42.3)
```

VV=ABS(-6.124) sets VV equal to 6.124.

```
>100 VV=ABS(-6.124)
```


ACCEPT

Format

ACCEPT [[AT(*row,column*)] [VALIDATE (*datatype* ,...)] [BEEP] [ERASE ALL] [SIZE(*numeric-expression*)] :] *variable*

Description

The ACCEPT statement suspends program execution until data is entered from the keyboard. Many options are available with ACCEPT, making it far more versatile than INPUT. It may accept data at any screen position, make an audible tone (beep) when ready to accept the data, erase all characters on the screen before accepting data, limit data accepted to a certain number of characters, and limit the type of characters accepted.

Options

The following options may appear in any order following ACCEPT.

AT(*row,column*) places the beginning of the input field at the specified *row* and *column*. *Rows* are numbered 1 through 24. *Columns* are numbered 1 through 28 with column 1 corresponding to what is called *column* 3 in the VCHAR, HCHAR, and GCHAR subprograms.

VALIDATE (*data-type* ,...) allows only certain characters to be entered. *Data-type* specifies which characters are acceptable. If more than one *data-type* is specified, a character from any of the *data-types* given is acceptable. The following are the *data-types*.

- UALPHA permits all uppercase alphabetic characters.
- DIGIT permits 0 through 9.
- NUMERIC permits 0 through 9, ".", "+", "-", and "E".
- *String-expression* permits the characters contained in *string-expression*.

BEEP sounds a short tone to signal that the computer is ready to accept input.

ERASE ALL fills the entire screen with the blank character before accepting input.

SIZE(*numeric-expression*) allows up to the absolute value of *numeric-expression* characters to be input. If *numeric-expression* is positive, the field in which the data is entered is cleared before input is accepted. If *numeric-expression* is negative, the input field is not blanked. This allows a default value to be previously placed in the field and entered by just pressing **ENTER**. If there is no SIZE clause, the line is blanked from the beginning position to the end of the line.

If the ACCEPT statement is used without the AT clause, the last two characters on the screen (at the lower right) are changed to "edge characters" (ASCII code 31).

Examples

ACCEPT AT(5,7):Y accepts data at the fifth row, seventh column of the screen into the variable Y.

```
>100 ACCEPT AT(5,7):Y
```

ACCEPT VALIDATE("YN"):R$ accepts Y or N into the variable R$.

```
>100 ACCEPT VALIDATE("YN"):R$
```

ACCEPT ERASE ALL:B accepts data into the variable B after putting the blank character into all screen positions.

```
>100 ACCEPT ERASE ALL:B
```

ACCEPT AT(R,C)SIZE(FIELDLEN) BEEP VALIDATE(DIGIT,"AYN"):X$ accepts a digit or the letters A, Y, or N into the variable X$. The length of the input may be up to FIELDLEN characters. The data is accepted at row R, column C, and a beep is sounded before data is accepted.

```
>100 ACCEPT AT(R,C)SIZE(FIELD
 LEN)BEEP VALIDATE(DIGIT,"AYN
 "):X$
```

Program

The program at the right illustrates a typical use of ACCEPT. It allows entry of up to 20 names and addresses, and then displays them all.

```
>100 DIM NAME$(20),ADDR$(20)
>110 DISPLAY AT(5,1)ERASE ALL
 :"NAME:"
>120 DISPLAY AT(7,1):"ADDRESS
 :"
>130 DISPLAY AT(23,1):"TYPE A
 ? TO END ENTRY."
>140 FOR S=1 TO 20
>150 ACCEPT AT(5,7)VALIDATE(U
 ALPHA,"?")BEEP SIZE(13):NAME
 $(S)
>160 IF NAME$(S)="?" THEN 200
>170 ACCEPT AT(7,10)SIZE(12):
 ADDR$(S)
>180 DISPLAY AT(7,10):"
 "
```



```
>190 NEXT S
>200 CALL CLEAR
>210 DISPLAY AT(1,1):"NAME","
 ADDRESS"
>220 FOR T=1 TO S-1
>230 DISPLAY AT(T+2,1):NAME$(
 T),ADDR$(T)
>240 NEXT T
>250 GOTO 250
```

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

ASC

Format

ASC(*string-expression*)

Description

The ASC function gives the ASCII character code which corresponds to the first character of *string-expression*. A list of the ASCII codes is given in *Appendix C*. The ASC function is the inverse of the CHR$ function.

Examples

PRINT ASC("A") prints 65.	`>100 PRINT ASC("A")`
B=ASC("1") sets B equal to 49.	`>100 B=ASC("1")`
DISPLAY ASC("HELLO") displays 72.	`>100 DISPLAY ASC("HELLO")`

ATN

Format

ATN(*numeric-expression*)

Description

The ATN function returns the measure of the angle (in radians) whose tangent is *numeric-expression*. If you want the equivalent angle in degrees, multiply by 180/PI. The value given by the ATN function is always in the range $-PI/2 < ATN(X) < PI/2$.

Examples

PRINT ATN(0) prints 0.	`>100 PRINT ATN(0)`
Q=ATN(.44) sets Q equal to .4145068746.	`>100 Q=ATN(.44)`

BREAK

Format

BREAK [*line-number-list*]

Description

The BREAK command requires a *line-number-list*. It causes the program to stop immediately before the lines in *line-number-list* are executed. After a breakpoint is taken because the line is listed in *line-number-list*, the breakpoint is removed and no more breakpoints occur at that line unless a new BREAK command or statement is given.

The BREAK statement without *line-number-list* causes the program to stop when it is encountered. The line at which the program stops is called a breakpoint. Every time a BREAK statement without *line-number-list* is encountered, the program stops even if an ON BREAK NEXT statement has been executed.

You can also cause a breakpoint in a program by pressing **SHIFT C** (CLEAR) while the program is running, unless breakpoints are being handled in some other way because of the action of ON BREAK.

BREAK is useful in finding out why a program is not running exactly as you expect it to. When the program has stopped you can print values of variables to find out what is happening in the program. You may enter any command or statement that can be used as a command. If you edit the program, however, you cannot resume with CONTINUE.

A way to remove breakpoints set with BREAK followed by line numbers is the UNBREAK command. Also, if a breakpoint is set at a program line and that line is deleted, the breakpoint is removed. Breakpoints are also removed when a program is saved with the SAVE command. See ON BREAK for a way to handle breakpoints.

Whenever a breakpoint occurs, the standard character set is restored. Thus any standard characters that had been redefined by CALL CHAR are restored to the standard characters. A breakpoint also restores the standard colors, deletes sprites, and resets sprite magnification to the default value of 1.

Options

The *line-number-list* is optional when BREAK is used as a statement, but is required when BREAK is used as a command. When present, it causes the program to stop immediately before the lines in *line-number-list* are executed. After a breakpoint is taken because the line is listed in *line-number-list*, the breakpoint is removed and no more breakpoints occur at that line unless a new BREAK command or statement is given.

Examples

BREAK as a statement causes a breakpoint when that statement is executed.	`>150 BREAK`
BREAK 120,130 as a statement causes breakpoints before execution of the line numbers listed.	`>110 BREAK 120,130`
BREAK 200,300,1105 as a command causes breakpoints before execution of the line numbers listed.	`>BREAK 200,300,1105`

BYE

Format

BYE

Description

The BYE command ends TI Extended BASIC and returns the computer to the master title screen. All open files are closed, all program lines are erased, and the computer is reset. Always use the BYE command instead of **SHIFT Q** (QUIT) to leave TI Extended BASIC. **SHIFT Q** (QUIT) does not close files, which may result in data being lost from external devices.

CALL

Format

CALL *subprogram-name* [(*parameter-list*)]

Description

The CALL statement transfers control to *subprogram-name*. The subprogram may be either one built into TI Extended BASIC, such as CLEAR, or one you have written. After the subprogram is executed, the next statement after the CALL statement is executed. CALL may be either a statement or a command for calling built-in TI Extended BASIC subprograms, but must be a statement when calling subprograms that you write.

Options

The *parameter-list* is defined according to the subprogram you are calling. Some require no parameters at all, some require parameters, and some have optional parameters. Each built-in subprogram is discussed under its own entry in this manual. The subprograms you can write are discussed in the section in Chapter II on subprograms and under SUB. The following are the *subprogram-names* of the built-in TI Extended BASIC subprograms.

CHAR	HCHAR	PATTERN
CHARPAT	INIT	PEEK
CHARSET	JOYST	POSITION
CLEAR	KEY	SAY
COINC	LINK	SCREEN
COLOR	LOAD	SOUND
DELSPRITE	LOCATE	SPGET
DISTANCE	MAGNIFY	SPRITE
ERR	MOTION	VCHAR
GCHAR		VERSION

Program

The program at the right illustrates the use of CALL with a supplied subprogram (CLEAR) in line 100 and the use of a written subprogram (TIMES) in line 120.

```
>100 CALL CLEAR
>110 X=4
>120 CALL TIMES(X)
>130 PRINT X
>140 STOP
>200 SUB TIMES(Z)
>210 Z=Z*PI
>220 SUBEND
>RUN
--screen clears
  12.56637061
```

CHAR subprogram

Format

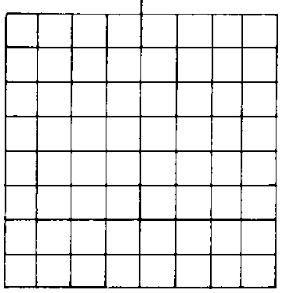
CALL CHAR(*character-code,pattern-identifier* [,...])

Description

The CHAR subprogram allows you to define special graphics characters. You can redefine the standard set of characters (ASCII codes 32-95) and the undefined characters, ASCII codes 96-143. *Note that fewer program defined characters are available in TI Extended BASIC than in TI BASIC, where ASCII codes 96-156 are allowed.* The CHAR subprogram is the inverse of the CHARPAT subprogram.

Character-code specifies the character which you wish to define and must be a numeric expression with a value from 32 through 143. *Pattern-identifier* is a 0 through 64 character string expression which specifies the pattern of the character(s) you are defining. This string expression is a coded representation of the dots which make up a character on the screen.

Each character is made up of 64 dots comprising an 8 by 8 grid as shown below.



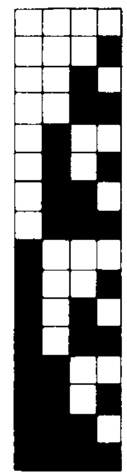
Each row is partitioned into two blocks of four dots each:

Each character in the *pattern-identifier* describes the pattern in one block of one row. The rows are described from left to right and from top to bottom. Therefore the first two characters in the *pattern-identifier* describe the pattern for row one of the grid, the next two the second row, and so on.

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

All the standard characters are set with the appropriate dots on. To create a new character, you specify what dots to turn on and leave off. In the computer a binary code, one number for each of the 64 dots, is used to specify which dots are on and off in a particular block. A more human-readable form of binary is hexadecimal. The following table shows all the possible on/off conditions for the four dots in a given block, and the binary and hexadecimal codes 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

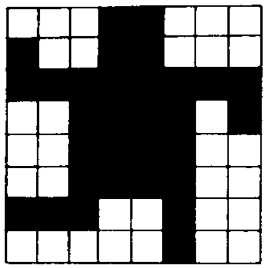
If the *pattern-identifier* is less than 16 characters, the computer assumes that the remaining characters are zeros. If the *pattern-identifier* is 17 to 32 characters, two *character-codes* are defined, the first with the first through sixteenth characters and the second with the remaining characters, with zeros added as needed. If the *pattern-identifier* is 33 to 48 characters, three *character-codes* are defined, the first with the first through sixteenth characters, the second with the seventeenth through thirty-second characters, and the third with the remaining characters, with zeros added as needed. If the *pattern-identifier* is 49 to 64 characters, four *character-codes* are defined, the first with the first through sixteenth characters, the second with the seventeenth through thirty-second characters, the third with the thirty-third through forty-eighth characters, and the fourth with the remaining characters, with zeros added as needed. If the *pattern-identifier* is longer than 64 characters or is long enough to define characters higher than character code 143, the excess is ignored.

Programs

To describe the dot pattern pictured below, you code this string for CALL CHAR:

"1898FF3D3C3CE404"

	LEFT BLOCKS	RIGHT BLOCKS	BLOCK CODES
ROW 1			18
ROW 2			98
ROW 3			FF
ROW 4			3D
ROW 5			3C
ROW 6			3C
ROW 7			E4
ROW 8			04



The program at the right uses this and one other string to make a figure "dance."

```
>100 CALL CLEAR
>110 A$="1898FF3D3C3CE404"
>120 B$="1819FFBC3C3C2720"
>130 CALL COLOR(9,7,12)
>140 CALL VCHAR(12,16,96)
>150 CALL CHAR(96,A$)
>160 GOSUB 200
>170 CALL CHAR(96,B$)
>180 GOSUB 200
>190 GOTO 150
>200 FOR DELAY=1 TO 50
>210 NEXT DELAY
>220 RETURN
>RUN
-- screen clears
-- character moves
 (Press SHIFT C to stop the
 program.)
```

If a program stops for a breakpoint, the predefined characters (ASCII codes 32 through 95) are reset to their standard pattern. Those with codes 96 through 143 keep their program defined pattern. When the program ends normally or because of an error, all predefined characters are reset.

```
>100 CALL CLEAR
>110 CALL CHAR(96,"FFFFFFFF
 FFFFFFFF")
>120 CALL CHAR(42,"0F0F0F0F0
 F0F0F0F")
>130 CALL HCHAR(12,17,42)
>140 CALL VCHAR(14,17,96)
>150 FOR DELAY=1 TO 500
>160 NEXT DELAY
>RUN
```


CHARPAT subprogram

CHAPTER 4

Format

CALL CHARPAT(*character-code,string-variable* [,...])

Description

The CHARPAT subprogram returns in *string-variable* the 16-character pattern identifier that specifies the pattern of *character-code*. The CHARPAT subprogram is the inverse of the CHAR subprogram. See the CHAR subprogram for an explanation of the value returned in *string-variable*.

Example

CALL CHARPAT(33,C$) sets C$ equal to "0010101010001000", the pattern identifier for character 33, the exclamation point.

```
>100 CALL CHARPAT(33,C$)
```

CHARSET subprogram

Format

CALL CHARSET

Description

The CHARSET subprogram restores the standard character patterns and standard colors for characters 32 through 95. Normally when a program is run by another program using RUN as a statement, characters 32 through 95 are not reset to their standard patterns and colors. CHARSET is useful when this feature is not desired.

Example

CALL CHARSET restores the standard characters and their colors.

```
>100 CALL CHARSET
```

CHR$

Format

CHR$(*numeric-expression*)

Description

The CHR$ function returns the character corresponding to the ASCII character code specified by *numeric-expression*. The CHR$ function is the inverse of the ASC function. A list of the ASCII character codes for each character in the standard character set is given in *Appendix C*.

Examples

PRINT CHR$(72) prints H.

```
>100 PRINT CHR$(72)
```

X$ = CHR$(33) sets X$ equal to !.

```
>100 X$=CHR$(33)
```

Program

For a complete list of all ASCII characters and their corresponding ASCII values, run the program on the right.

```
>100 CALL CLEAR
>110 FOR A=32 TO 95
>120 PRINT A;"   ";CHR$(A);"
 ";
>130 NEXT A
```


CLEAR subprogram

Format

CALL CLEAR

Description

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

Programs

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

```
>100 CALL CLEAR
>110 PRINT "HELLO THERE!"
>120 PRINT "HOW ARE YOU?"
>RUN
--screen clears
 HELLO THERE!
 HOW ARE YOU?
```

If the space character (ASCII code 32) has been redefined by the CALL CHAR subprogram, the screen is filled with the new character when CALL CLEAR is performed.

```
>100 CALL CHAR(32,"0103070F1F
 3F7FFF")
>110 CALL CLEAR
>120 GOTO 120
>RUN
--screen is filled with ◢
 (Press SHIFT C to stop the
 program.)
```

CLOSE

Format

CLOSE *#file-number* [:DELETE]

Description

The CLOSE statement stops a program's use of the file referenced by *#file-number*. After the CLOSE statement is performed, the file cannot be used by the program unless you OPEN it again. The computer no longer associates the *#file-number* with the closed file, so you can assign that number to another file.

When no program is running, the following actions close all open files:

- Editing the program
- Entering the BYE command
- Entering the RUN command
- Entering the NEW command
- Entering the OLD command
- Entering the SAVE command
- Entering the LIST command to a device

If you use **SHIFT Q** (QUIT) to leave TI Extended BASIC, the computer does not close any open files, and you may lose data on any files that are open. To avoid this possibility, you should leave TI Extended BASIC with BYE instead of **SHIFT Q** (QUIT).

Options

You may delete a diskette file at the same time you close it by adding ":DELETE" to the statement. Other devices, such as cassette recorders, do not allow DELETE. The manual for each device discusses the use of DELETE.

Examples

When the computer performs the CLOSE statement for a cassette tape recorder, you receive instructions for operating the recorder.

```
>100 OPEN #24:"CS1",INTERNAL,
 INPUT,FIXED
 .
 .
 .
--program lines
 .
 .
 .
>200 CLOSE #24
>RUN
--opening instructions
 .
 .
 .
--program runs
 .
 .
 .
* PRESS CASSETTE STOP     CS1
THEN PRESS ENTER
```

The CLOSE statement for a diskette requires no further action on your part.

```
>100 OPEN #24:"DSK1.MYDATA",I
 NTERNAL,INPUT,FIXED
 .
 .
 .
--program lines
 .
 .
 .
>200 CLOSE #24
>RUN
--program runs
```

COINC subprogram

Format

CALL COINC(*#sprite-number,#sprite-number,tolerance,numeric-variable*)
CALL COINC(*#sprite-number,dot-row,dot-column,tolerance,numeric-variable*)
CALL COINC(ALL,*numeric-variable*)

Description

The COINC subprogram detects a coincidence between a sprite and another sprite or a position on the screen. The value returned in *numeric-variable* is −1 if there is a coincidence and 0 if there is no coincidence.

If the keyword ALL is given, the coincidence of any two sprites is reported. If two sprites are identified by *#sprite-number*, their coincidence is reported. If *#sprite-number* and a location are identified, their coincidence is reported.

If the keyword ALL is given, sprites are coincident only if one or more of the dots which make them up occupy the same position on the screen. If two sprites or a sprite and a location are given, then *tolerance* must be specified, and two sprites are coincident if their upper left hand corners are within the value specified by *tolerance*. A sprite and a location are coincident if the upper left hand corner of the sprite and the position specified by *dot-row* and *dot-column* are within the value specified by *tolerance*. These coincidents are reported even if there is no apparent overlap of the sprites or the sprite and the position.

Dot-row and *dot-column* are numbered consecutively starting with 1 in the upper left hand corner of the screen. Thus the *dot-row* can be from 1 to 192 and the *dot-column* can be from 1 to 256. (Actually the *dot-row* can go up to 256, but the positions from 193 through 256 are off the bottom of the screen.) If any part of the sprite occupies the position given, then there is a coincidence.

Whether or not a coincidence is detected depends on several variables. If the sprites are moving very quickly, COINC may not be able to detect their coincidence. Also, COINC checks for a coincidence only when it is called, so a program may miss a coincidence that occurs when the program is executing some other statement.

Program

The program at the right defines two sprites that consist of a triangle.

```
>100 CALL CLEAR
>110 S$="0103070F1F3F7FFF"
>120 CALL CHAR(96,S$)
>130 CALL CHAR(100,S$)
>140 CALL SPRITE(#1,96,7,8,8)
>150 CALL SPRITE(#2,100,5,1,1)
```

Line 160 shows a coincidence because the sprites are within 10 dots of each other.

```
>160 CALL COINC(#1,#2,10,C)
>170 PRINT C
```

Line 180 shows no coincidence because the shaded areas of the sprites are not coincident.

```
>180 CALL COINC(ALL,C)
>190 PRINT C
>RUN
 -1
  0
```

COLOR subprogram

Format

CALL COLOR(#*sprite-number*,*foreground-color* [,...])
CALL COLOR(*character-set*,*foreground-color*,*background-color* [,...])

Description

The COLOR subprogram allows you to specify either a *foreground-color* for #*sprite-number* or a *foreground-color* and *background-color* for characters in the *character-set*. In a given CALL COLOR, you may define sprite color(s) or character set colors, but not both.

Each character 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*. In sprites, the *background-color* is always code 1, transparent, which allows characters and the screen color to show through. To change the screen color, see the SCREEN subprogram. *Foreground-color* and *background-color* must have values from 1 through 16. The color codes are shown 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

Until CALL COLOR is performed, the standard *foreground-color* is black (code 2) and the standard *background-color* is transparent (code 1) for all characters. Sprites have their color assigned when they are created. When a breakpoint occurs, all characters are reset to the standard colors.

To use CALL COLOR you must also specify to which of the fifteen character sets the character belongs. (*Note that TI BASIC has sixteen character sets while TI Extended BASIC has fifteen.*) The list of ASCII character codes for the standard characters is given in Appendix C. The *character-set* numbers are given below:

Set Number	*Character Codes*
0	30-31
1	32-39
2	40-47
3	48-55
4	56-63
5	64-71
6	72-79
7	80-87
8	88-95
9	96-103
10	104-111
11	112-119
12	120-127
13	128-135
14	136-143

Examples

CALL COLOR(3,5,8) sets the *foreground-color* of characters 48 through 55 to 5 (dark blue) and the *background-color* to 8 (cyan).

```
>100 CALL COLOR(3,5,8)
```

CALL COLOR(#5,16) sets sprite number 5 to have a *foreground-color* of 16 (white). The *background-color* is always 1 (transparent).

```
>100 CALL COLOR(#5,16)
```

CALL COLOR(#7,INT(RND*16+1)) sets sprite number 7 to have a *foreground-color* chosen randomly from the 16 colors available. The *background-color* is 1 (transparent).

```
>100 CALL COLOR(#7,INT(RND*16
 +1))
```

CONTINUE

Format

CONTINUE
CON

Description

The CONTINUE command restarts a program which has been stopped by a breakpoint. It may be entered whenever a program has stopped running because of a breakpoint caused by the BREAK command or statement or **SHIFT C** (CLEAR). However, you cannot use the CONTINUE command if you have edited a program line. CONTINUE may be abbreviated as CON.

When a breakpoint occurs, the standard character set and standard colors are restored. Sprites cease to exist. CONTINUE does not restore standard characters that have been reset or any colors. Otherwise, the program continues as if no breakpoint had occurred.

COS

Format

COS(*radian-expression*)

Description

The cosine function gives the trigonometric cosine of *radian-expression*. If the angle is in degrees, multiply the number of degrees by PI/180 to get the equivalent angle in radians.

Program

The program on the right gives the cosine of several angles.

```
>100 A=1.047197551196
>110 B=60
>120 C=45*PI/180
>130 PRINT COS(A);COS(B)
>140 PRINT COS(B*PI/180)
>150 PRINT COS(C)
>RUN
 .5 -.9524129804
 .5
 .7071067812
```

DATA

Format

DATA *data-list*

Description

The DATA statement allows you to store data inside your program. The data, which may be numeric or string constants, is listed in *data-list* separated by commas. During program execution, the READ statement assigns the values in *data-list* to the variables specified in *variable-list* in the READ statement.

DATA statements may be located anywhere in a program. However, the order in which they appear is important. Data from several DATA statements is read sequentially, beginning with the first item in the first DATA statement. If a program has more than one DATA statement, the DATA statements are read in the order in which they appear in the program, unless otherwise specified by a RESTORE statement. Thus the order in which data appears in the program normally determines the order in which data is read. DATA statements cannot be part of multiple statement lines.

Data in *data-list* must correspond to the type of the variable to which it is assigned in the READ statement. Thus if a numeric variable is specified in the READ statement, a numeric constant must be in the corresponding position in the DATA statement. Similarly, if a string variable is specified, a string constant must be supplied. A number is a valid string, so you may have a numeric constant in a DATA statement where a string is called for in the READ statement. If a DATA statement contains adjacent commas, the computer assumes you want to enter a null string (a string with no characters).

When using string constants in a DATA statement, you may enclose the string in quotes. However, if the string you include contains a comma, leading spaces, or trailing spaces, you *must* enclose the string in quotes. If the string is enclosed in quotes, quotes in the string are represented by double quotes.

Program

The program at the right reads and prints several numeric and string constants. Lines 100 through 130 read five sets of data and print their values, two to a line.

```
>100 FOR A=1 TO 5
>110 READ B,C
>120 PRINT B;C
>130 NEXT A
>140 DATA 2,4,6,7,8
>150 DATA 1,2,3,4,5
>160 DATA """THIS HAS QUOTES"
 ""
>170 DATA " NO QUOTES, HERE"
>180 DATA NO QUOTES HERE EITH
 ER
```

Lines 190 through 220 read seven data elements and print each on its own line.

```
>190 FOR A=1 TO 7
>200 READ B$
>210 PRINT B$
>220 NEXT A
>230 DATA 1,NUMBER,,TI
>RUN
```

First two elements of line 140.	2 4
Second two elements of line 140.	6 7
Last element of line 140 and first of line 150	8 1
Second and third elements of line 150.	2 3
Fourth and fifth elements of line 150.	4 5
Line 160.	"THIS HAS QUOTES"
Line 170.	NO QUOTES, HERE
Line 180.	NO QUOTES HERE EITHER
First element of line 230.	1
Second element of line 230.	NUMBER
Null string for two commas in line 230.	
Last element of line 230.	TI

DEF

Format

DEF *function-name* [(*parameter*)] = *expression*

Description

The DEF statement allows you to define your own functions. *Function-name* may be any variable name. If you specify a *parameter* following *function-name*, the *parameter* must be enclosed in parentheses and may be any scalar variable name. If *expression* is a string, *function-name* must be a string variable name, i.e. the last character must be a dollar sign.

The DEF statement must occur at a lower numbered line than any reference to the function it defines. However, a DEF statement may not appear in an IF-THEN-ELSE statement. When the computer encounters a DEF statement during program execution, it proceeds to the next statement without taking any action. A function may be used in any string or numeric expression by using *function-name* followed by an expression enclosed in parentheses if a *parameter* was specified in the DEF statement.

When a reference to the function is encountered in an expression (by using *function-name* in a statement), the function is evaluated using the current values of the variables specified in the DEF statement and the value of *parameter* if there is one. A DEF statement can refer to other defined functions. However, the function you specify may not refer to itself either directly (e.g. DEF B=B*2) or indirectly (e.g. DEF F=G::DEF G=F).

Attempting to print the value of a function with PRINT used as a command does not work if the Memory Expansion is connected to your computer.

Options

If you specify a *parameter* for a function, when a reference to the function is encountered in an expression, its value is assigned to *parameter*. The value of the function is then determined using the value of *parameter* and the values of the other variables in the DEF statement. If *parameter* is given in the DEF statement, an argument value must always be given when referring to the function.

The *parameter* name used in the DEF statement affects only the DEF statement in which it is used. This means that it is distinct from any other variable with the same name which appears elsewhere in the program.

Parameter may not be used as an array. You can use an array element in a function as long as the array does not have the same name as parameter. For example you may use DEF F(A)=B(Z) but not DEF F(A)=A(Z).

Examples

DEF PAY(OT)=40*RATE+1.5*RATE*OT defines PAY so that each time it is encountered in a program the pay is figured using the RATE of pay times 40 plus 1.5 times the rate of pay times the overtime hours.

```
>100 DEF PAY(OT)=40*RATE+1.5*
 RATE*OT
```

DEF RND20=INT(RND*20+1) defines RND20 so that each time it is encountered in a program an integer from 1 through 20 is given.

```
>100 DEF RND20=INT(RND*20+1)
```

DEF FIRSTWORD$(NAME$)=SEG$(NAME$,1,POS(NAME$," ",1)−1) defines FIRSTWORD$ to be the part of NAME$ that preceeds a space.

```
>100 DEF FIRSTWORD$(NAME$)=SE
 G$(NAME$,1,POS(NAME$," ",1)-
 1)
```

DELETE

Format

DELETE *device-filename*

Description

The DELETE command allows you to remove a program or data file from the computer's filing system. *Device-filename* is a string expression. If a string constant is used, it must be enclosed in quotes. You may also delete data files by using the keyword DELETE in the CLOSE statement.

Some devices (such as diskettes) allow deleting files; others (such as cassettes) do not. Read the manual for the specific device for more information.

Example

DELETE "DSK1.MYFILE" deletes the file named MYFILE from the diskette in disk drive 1.

```
>DELETE "DSK1.MYFILE"
```

Program

The program on the right illustrates a use of DELETE.

```
>100 INPUT "FILENAME: ":X$
>110 DELETE X$
```


DELSPRITE subprogram

Format

CALL DELSPRITE(*#sprite-number* [,...])
CALL DELSPRITE(ALL)

Description

The DELSPRITE subprogram removes sprites from further access by a program. You may delete one or more sprites by specifying their numbers preceded by a number sign (#) and separated by commas, or you may delete all sprites by specifying ALL. After being deleted with DELSPRITE, a sprite can be recreated with the SPRITE subprogram.

Examples

CALL DELSPRITE(#3) deletes sprite number 3.

```
>100 CALL DELSPRITE(#3)
```

CALL DELSPRITE(#4,#3*C) deletes sprite number 4 and the sprite whose number is found by multiplying 3 by C.

```
>100 CALL DELSPRITE(#4,#3*C)
```

CALL DELSPRITE(ALL) deletes all sprites.

```
>100 CALL DELSPRITE(ALL)
```

DIM

Format

DIM *array-name(integer1* [,*integer2*] ... [,*integer7*] [,...])

Description

The DIM statement reserves space in the computer's memory for numeric and string arrays. You can dimension an array only once in a program. If you dimension an array, the DIM statement must appear in the program at a lower numbered line than any other reference to the array. If you dimension more than one array in a single DIM statement, *array-names* are separated by commas. *Array-name* may be any variable name. A DIM statement may not appear in an IF-THEN-ELSE statement.

You may have up to seven-dimensional arrays in TI Extended BASIC. The number of *integers* separated by commas following the array name determines how many dimensions the array has. The values of the integers determine the number of elements in each dimension.

Space is allocated for an array after you enter the RUN command but before the first statement is executed. Each element in a string array is a null string and each element in a numeric array is zero until it is replaced with another value.

The values of the *integers* determine the maximum value of each subscript for that array. If you are using an array not defined in a DIM statement, the maximum value of each subscript is 10. The first element is zero unless an OPTION BASE statement sets the minimum subscript value to 1. Thus an array defined as DIM A(6) is a one dimensional array with seven elements unless the zero subscript is eliminated by the OPTION BASE statement.

Examples

DIM X$(30) reserves space in the computer's memory for 31 members of the array called X$.

```
>100 DIM X$(30)
```

DIM D(100),B(10,9) reserves space in the computer's memory for 101 members of the array called D and 110 (11 times 10) members of the array called B.

```
>100 DIM D(100),B(10,9)
```


DISPLAY

Format

DISPLAY [[AT(*row,column*)] [BEEP] [ERASE ALL] [SIZE(*numeric-expression*)] :] *variable-list*

Description

The DISPLAY statement displays information on the screen. Many options are available with DISPLAY, making it far more versatile than PRINT. It may display data at any screen position, make an audible tone (beep) when displaying data, blank screen positions, and erase all characters on the screen before displaying data.

Options

AT(*row,column*) places the beginning of the display field at the specified *row* and *column*. *Rows* are numbered 1 through 24. *Columns* are numbered 1 through 28 with *column* 1 corresponding with what is called *column* 3 in the VCHAR, HCHAR, and GCHAR subprograms. If the AT option is not present, data is displayed at *row* 24, *column* 1, just as it is with the PRINT statement.

BEEP sounds a short tone when the data is displayed.

ERASE ALL fills the entire screen with the blank character before displaying data.

SIZE(*numeric-expression*) puts *numeric-expression* blank characters on the screen starting at *row* and *column*. If the SIZE option is not present, the rest of the row at which data is to be displayed is blanked. If *numeric-expression* is larger than the number of positions remaining in the row, only the rest of the row is blanked.

Examples

DISPLAY AT(5,7):Y displays the value of Y at the fifth row, seventh column of the screen.

```
>100 DISPLAY AT(5,7):Y
```

DISPLAY ERASE ALL:B puts the blank character into all screen positions before displaying the value of B.

```
>100 DISPLAY ERASE ALL:B
```

DISPLAY AT(R,C) SIZE(FIELDLEN) BEEP:X$ displays the value of X$ at row R, column C. First it beeps and blanks FIELDLEN characters.

```
>100 DISPLAY AT(R,C) SIZE(FIE
 LDLEN)BEEP:X$
```

Program

The program at the right illustrates a use of DISPLAY. It allows you to position blocks at any screen position to draw a figure or design.

```
>100 CALL CLEAR
>110 CALL COLOR(9,5,5)
>120 DISPLAY AT(23,1):"ENTER
 ROW AND COLUMN."
>130 DISPLAY AT (24,1):"ROW:
    COLUMN:"
>140 FOR COUNT=1 TO 2
>150 CALL KEY(0,ROW(COUNT),S)
>160 IF S<=0 THEN 150
>170 DISPLAY AT(24,5+COUNT)SI
 ZE(1):STR$(ROW(COUNT)-48)
>180 NEXT COUNT
>190 FOR COUNT=1 TO 2
>200 CALL KEY(0,COLUMN(COUNT)
 ,S)
>210 IF S<=0 THEN 200
>220 DISPLAY AT(24,16+COUNT)S
 IZE(1):STR$(COLUMN(COUNT)-48
 )
>230 NEXT COUNT
>240 ROW1=10*(ROW(1)-48)+ROW(
 2)-48
>250 COLUMN1=10*(COLUMN(1)-48
 )+COLUMN(2)-48
>260 DISPLAY AT(ROW1,COLUMN1)
 SIZE(1):CHR$(96)
>270 GOTO 130
```

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

DISPLAY USING

Format

DISPLAY [*option-list*:] USING *string-expression* [: *variable-list*]
DISPLAY [*option-list*:] USING *line-number* [: *variable-list*]

Description

The DISPLAY...USING statement is the same as DISPLAY with the addition of the USING clause, which specifies the format of the data in *variable-list*. If *string-expression* is present, it defines the format. If *line-number* is present, it refers to the line number of an IMAGE statement. See IMAGE for an explanation of how the format is defined.

Examples

DISPLAY AT(10,4):USING "##.##":N displays the value of N at the tenth row and fourth column, with the format "##.##".

```
>100 DISPLAY AT(10,4):USING
 "##.##":N
```

DISPLAY USING "##.##":N displays the value of N at the 24th row and first column, with the format "##.##".

```
>100 DISPLAY USING "##.##":N
```

DISTANCE subprogram

Format

CALL DISTANCE(*#sprite-number,#sprite-number,numeric-variable*)
CALL DISTANCE(*#sprite-number,dot-row,dot-column,numeric-variable*)

Description

The DISTANCE subprogram returns the square of the distance between two sprites or between a sprite and a location. The position of each sprite is considered to be its upper left hand corner. *Dot-row* and *dot-column* are from 1 to 256. The squared distance is returned in *numeric-variable*.

The number returned is computed as follows: The difference between the *dot-rows* of the sprites (or the sprite and the location) is found and squared. Then the difference between the *dot-columns* of the sprites (or the sprite and the location) is found and squared. Then the two squares are added. If the sum is larger than 32767, then 32767 is returned. The distance between the sprites (or the sprite and the location) is the square root of the value returned.

Examples

CALL DISTANCE(#3,#4,DIST) sets DIST equal to the square of the distance between the upper left hand corners of sprite #3 and sprite #4.

```
>100 CALL DISTANCE(#3,#4,DIST)
```

CALL DISTANCE(#4,18,89,D) sets D equal to the square of the distance between the upper left hand corner of sprite #4 and position 18, 89.

```
>100 CALL DISTANCE(#4,18,89,D)
```


END

Format

END

Description

The END statement ends your program and may be used interchangeably with the STOP statement. Although the END statement may appear anywhere, it is normally placed as the last line in a program and thus ends the program both physically and logically. The STOP statement is usually used in other places that you want your program to halt. In TI Extended BASIC you are not required to use the END statement. The program automatically stops after it executes the highest numbered line.

EOF

Format

EOF(*file-number*)

Description

The EOF function is used to test whether there is another record to be read from a file. The value of *file-number* indicates the file to be tested and must correspond to the number of an open file. The EOF function cannot be used with cassettes.

The EOF function always assumes that the next record is going to be read sequentially, even if you are using a RELATIVE file.

The value that the EOF function provides depends on where you are in the file. If you are not at the last record of the file, the function returns a value of 0. If you are at the last record of the file, the function returns a value of 1. If the diskette or other storage medium is full, you are at the end of the file, and there is no more room for any data, the function returns a value of – 1.

For more information, see the Disk Memory System manual.

Examples

PRINT EOF(3) prints a value according to whether you are at the end of the file that was opened as #3.

```
>100 PRINT EOF(3)
```

IF EOF(27)< >0 THEN 1150 transfers control to line 1150 if you are at the end of the file that was opened as #27.

```
>100 IF EOF(27)<>0 THEN 1150
```

IF EOF(27) THEN 1150 transfers control to line 1150 if you are at the end of the file that was opened as #27.

```
>100 IF EOF(27) THEN 1150
```


ERR subprogram

Format

CALL ERR(*error-code*,*error-type* [,*error-severity*,*line-number*])

Description

The ERR subprogram returns the *error-code* and *error-type* of the most recent uncleared error. An error is cleared when it has been accessed by the ERR subprogram, another error has occured, or the program has ended.

Error-codes are two or three digit numbers. The meanings of each of the codes is in *Appendix N*.

If *error-type* is a negative number, then the error was in the execution of the program. If the *error-code* is 130 (I/O ERROR), the *error-type* is a positive number and the number is the number of the file that caused the error.

If no error has occured, CALL ERR returns all values as zeros.

CALL ERR is used in conjunction with ON ERROR.

Options

You may optionally obtain the *error-severity* and *line-number* on which the error occured. The *error-severity* is always 9. The *line-number* is the number of the line being executed when the error occurred. It is not always the line that is the source of the problem since an error may occur because of values generated or actions taken elsewhere in a program.

Examples

CALL ERR(A,B) sets A equal to the *error-code* and B equal to the *error-type* of the most recent error.

```
>100 CALL ERR(A,B)
```

CALL ERR(W,X,Y,Z) sets W equal to the *error-code*, X equal to the *error-type*, Y equal to the *error-severity*, and Z equal to the *line-number* of the most recent error.

```
>100 CALL ERR(W,X,Y,Z)
```

ERR SUBPROGRAM

Program

The program on the right illustrates the use of CALL ERR. An error is caused in line 110 by calling for an illegal screen color. Because of line 100, control is transfered to line 130. Line 140 prints the values obtained. The 79 indicates that a bad value was provided.
The – 1 indicates that the error was in a statement. The 9 is the *error-severity*. The 110 indicates that the error occured in line 110.

```
>100 ON ERROR 130
>110 CALL SCREEN (18)
>120 STOP
>130 CALL ERR(W,X,Y,Z)
>140 PRINT W;X;Y;Z
>RUN
>  79 -1  9  110
```


EXP

Format

EXP(*numeric-expression*)

Description

The EXP function returns the exponential value (e^X) of *numeric-expression*. The value of e is 2.718281828459.

Examples

Y = EXP(7) assigns to Y the value of e raised to the seventh power which is 1096.633158429.

```
>100 Y=EXP(7)
```

L = EXP(4.394960467) assigns to L the value of e raised to the 4.394960467 power which is 81.04142688868.

```
>100 L=EXP(4.394960467)
```

FOR TO [STEP]

Format

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

Description

The FOR-TO-STEP statement repeats execution of the statements between FOR-TO-STEP and NEXT until the *control-variable* is outside the range of *initial-value* to *limit*. The FOR-TO-STEP statement is useful when repeating the same steps in a loop. The FOR-TO-STEP statement cannot be used in an IF-THEN-ELSE statement.

Control-variable may be any unsubscripted numeric variable. It acts as a counter for the loop. *Initial-value* and *limit* are numeric expressions. The loop starts with *control-variable* given a value of *initial-value*. The second time through the loop, the value of *control-variable* is changed by one or optionally by *increment*, which may be a positive or negative number. This continues until the value of *control-variable* is outside the range *initial-value* to *limit*. Then the statement after NEXT is executed. The value of *control-variable* is not changed when the computer leaves the loop.

The value of *control-variable* can be changed within the loop, but this must be done carefully to avoid unexpected results. Loops may be "nested," that is one loop may be contained wholly within another. You may leave a loop using GOTO, GOSUB, IF-THEN-ELSE, or the like, and then return. However, you may not enter a FOR-NEXT loop at any point except at its start.

If *initial-value* exceeds *limit* at the beginning of the FOR-NEXT loop, none of the statements in the loop are executed. Instead execution continues with the first statement after the NEXT statement.

Examples

FOR A = 1 TO 5 STEP 2 executes the statements between this FOR and NEXT A three times, with A having values of 1, 3, and 5. After the loop is finished, A has a value of 7.

```
>100 FOR A=1 TO 5 STEP 2
```

FOR J = 7 TO −5 STEP −.5 executes the statements between this FOR and NEXT J 25 times, with J having values of 7, 6.5, 6, ..., −4, −4.5, and −5. After the loop is finished, J has a value of −5.5.

```
>100 FOR J=7 TO -5 STEP -.5
```


Program

The program at the right illustrates a use of the FOR-TO-STEP statement. There are three FOR-NEXT loops, with *control-variables* of CHAR, ROW, and COLUMN.

```
>100 CALL CLEAR
>110 D=0
>120 FOR CHAR=33 TO 63 STEP 3
 0
>130 FOR ROW=1+D TO 21+D STEP
 4
>140 FOR COLUMN=1-D TO 29+D S
 TEP 4
>150 CALL VCHAR(ROW,COLUMN,CH
 AR)
>160 NEXT COLUMN
>170 NEXT ROW
>180 D=2
>190 NEXT CHAR
>200 GOTO 200
 (Press SHIFT C to stop the
 program.)
```

GCHAR subprogram

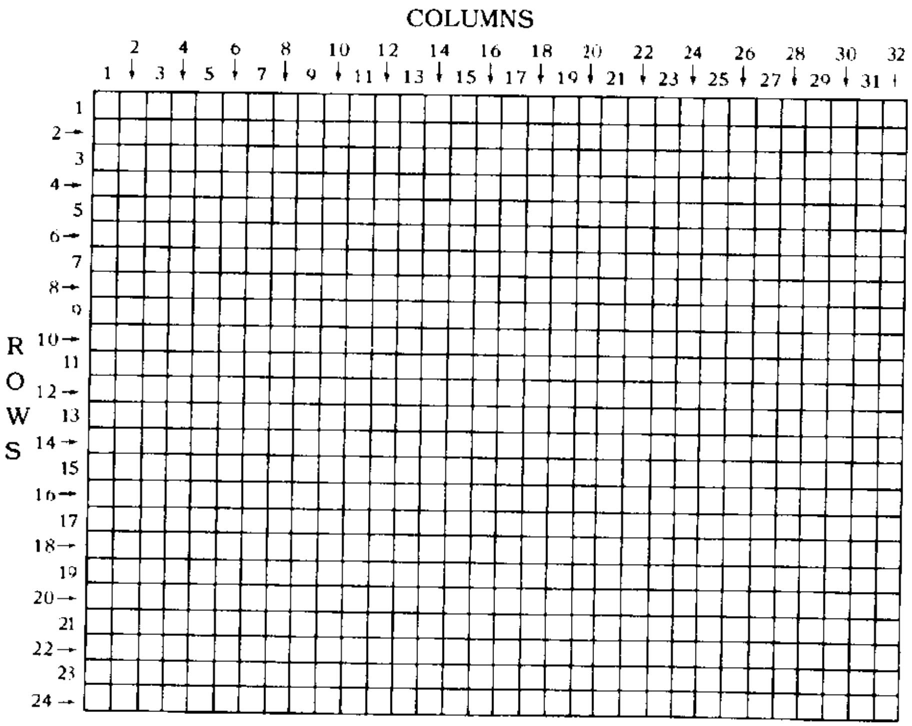
Format

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

Description

The GCHAR subprogram reads a character from anywhere on the display screen. The computer returns in *numeric-variable* the ASCII code for the character in the position described by *row* and *column*.

Row and *column* are numeric expressions. A value of 1 for *row* indicates the top of the screen. A value of 1 for the *column* indicates the left side of the screen. The screen can be thought of as a grid as shown below.



Examples

CALL GCHAR(12,16,X) assigns to X the ASCII code of the character that is in row 12, column 16.	`>100 CALL GCHAR(12,16,X)`
CALL GCHAR(R,C,K) puts into K the ASCII code of the character that is in row R, column C.	`>100 CALL GCHAR(R,C,K)`

GOSUB

Format

GOSUB *line-number*
GO SUB *line-number*

Description

The GOSUB statement allows transfer to a subroutine. When executed, control is transferred to *line-number* and that statement and any following (which may include any statements, including GOTO statements and other GOSUB statements) are executed. When a RETURN statement is encountered, control is returned to the next statement following the GOSUB statement. Subroutines are most useful when the same action is to be performed in different parts of a program. See also ON...GOSUB. Subroutines in TI Extended BASIC may call themselves.

Example

GOSUB 200 transfers control to statement 200. That statement and the ones up to RETURN are executed, and then control returns to the statement after the calling statement.	`>100 GOSUB 200`

GOSUB

Program

The program on the right illustrates a use of GOSUB. The subroutine at line 260 figures the factorial of the value of NUMB. The whole program figures the solution to the equation

$$NUMB = \frac{X!}{Y! * (X-Y)!}$$

where the exclamation point means factorial. This formula is used to figure certain probabilities. For instance, if you enter X as 52 and Y as 5, you'll find the number of possible five card poker hands.

```
>100 CALL CLEAR
>110 INPUT "ENTER X AND Y: ":
 X,Y
>120 IF X<Y THEN 110
>130 IF X>69 OR Y>69 THEN 110
>140 NUMB=X
>150 GOSUB 260
>160 NUMERATOR=NUMB
>170 NUMB=Y
>180 GOSUB 260
>190 DENOMINATOR=NUMB
>200 NUMB=X-Y
>210 GOSUB 260
>220 DENOMINATOR=DENOMINATOR*
 NUMB
>230 NUMB=NUMERATOR/DENOMINAT
 OR
>240 PRINT "NUMBER IS";NUMB
>250 STOP
>260 REM FIGURE FACTORIAL
>270 IF NUMB<0 THEN PRINT "NE
 GATIVE" :: GOTO 110
>280 IF NUMB<2 THEN NUMB=1 ::
 GOTO 330
>290 MULT=NUMB-1
>300 NUMB=NUMB*MULT
>310 MULT=MULT-1
>320 IF MULT>1 THEN 300
>330 RETURN
```


GOTO

CHAPTER 4

Format

GOTO *line-number*

GO TO *line-number*

Description

The GOTO statement allows you to transfer control unconditionally to another line within a program. When a GOTO statement is executed, control is passed to the first statement on the line specified by *line-number*.

The GOTO statement should not be used to transfer control into subprograms.

Program

The program at the right shows the use of GOTO in line 160. Anytime that line is reached the program executes line 130 next and proceeds from that new point.

```
>100 REM ADD 1 THROUGH 100
>110 ANSWER=0
>120 NUMB=1
>130 ANSWER=ANSWER+NUMB
>140 NUMB=NUMB+1
>150 IF NUMB>100 THEN 170
>160 GOTO 130
>170 PRINT "THE ANSWER IS";AN
 SWER
>RUN
THE ANSWER IS 5050
```

HCHAR subprogram

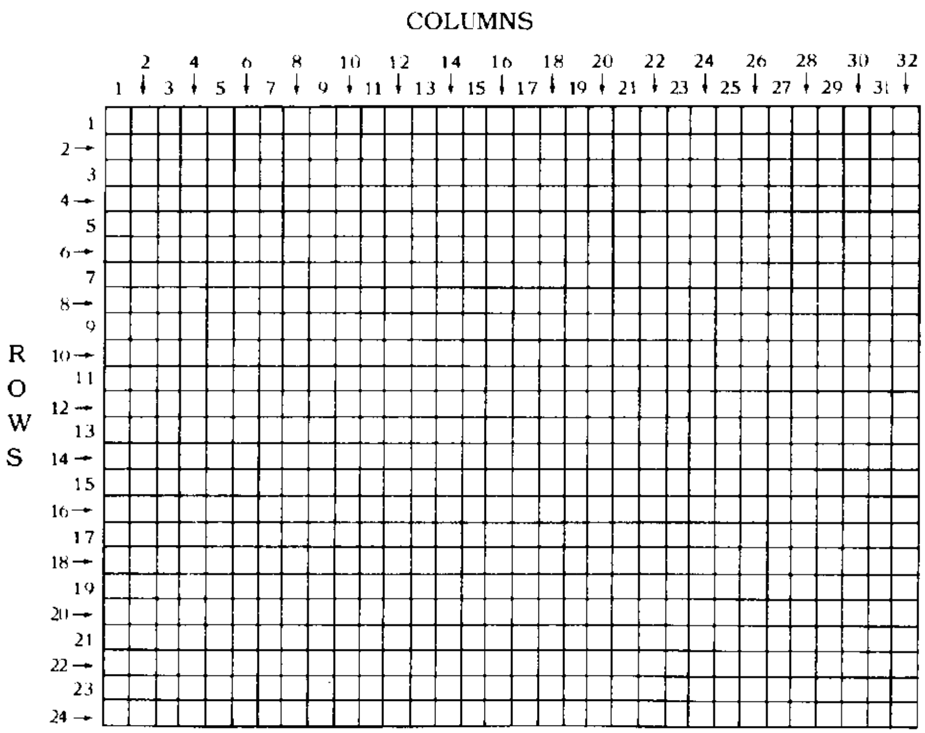
Format

CALL HCHAR(*row*,*column*,*character-code* [,*repetition*])

Description

The HCHAR subprogram displays a character anywhere on the display screen and optionally repeats it horizontally. The character with the ASCII value of *character-code* is placed in the position described by *row* and column and is repeated horizontally *repetition* times.

A value of 1 for *row* indicates the top of the screen. A value of 24 is the bottom of the screen. A value of 1 for *column* indicates the left side of the screen. A value of 32 is the right side of the screen. The screen can be thought of as a grid as shown below.



Examples

CALL HCHAR(12,16,33) places character 33 (an exclamation point) in row 12, column 16.

```
>100 CALL HCHAR(12,16,33)
```

CALL HCHAR(1,1,ASC("!"),768) places an exclamation point in row 1, column 1, and repeats it 768 times, which fills the screen.

```
>100 CALL HCHAR(1,1,ASC("!"),
 768)
```

CALL HCHAR(R,C,K,T) places the character with an ASCII code specified by the value of K in row R, column C and repeats it T times.

```
>100 CALL HCHAR(R,C,K,T)
```

IF THEN [ELSE]

Format

IF *relational-expression* THEN *line-number1* [ELSE *line-number2*]
IF *relational-expression* THEN *statement1* [ELSE *statement2*]
IF *numeric-expression* THEN *line-number1* [ELSE *line-number2*]
IF *numeric-expression* THEN *statement1* [ELSE *statement2*]

Description

The IF-THEN-ELSE statement allows you to transfer control to *line-number1* or to perform *statement1* if *relational-expression* is true or if *numeric-expression* is not equal to zero. Otherwise control passes to the next statement, or optionally to *line-number2* or *statement2*.

Statement1 and *statement2* may each be several statements long, separated by the statement separator symbol. They are only executed if the clause immediately before them is executed. The IF-THEN-ELSE statement cannot contain DATA, DEF, DIM, FOR, NEXT, OPTION BASE, SUB, or SUBEND.

Examples

IF X>5 THEN GOSUB 300 ELSE X=X+5 operates as follows: If X is greater than 5, then GOSUB 300 is executed. When the subroutine is ended, control returns to the line following this line. If X is 5 or less, X is set equal to X+5 and control passes to the next line.

```
>100 IF X>5 THEN GOSUB 300 EL
 SE X=X+5
```

IF Q THEN C=C+1::GOTO 500::ELSE L=L/C::GOTO 300 operates as follows: If Q is not zero, then C is set equal to C+1 and control is transferred to line 500. If Q is zero, then L is set equal to L/C and control is transferred to line 300.

```
>100 IF Q THEN C=C+1::GOTO 50
 0::ELSE L=L/C::GOTO 300
```

IF A>3 THEN 300 ELSE A=0:: GOTO 10 operates as follows: If A is greater than 3, then control is transferred to line 300. Otherwise, A is reset to zero and control is transferred to line 10.

```
>100 IF A>3 THEN 300 ELSE A=0
 ::GOTO 10
```


IF A$="Y" THEN COUNT=COUNT+1::DISPLAY AT(24,1):"HERE WE GO AGAIN!"::GOTO 300 operates as follows: If A$ is not equal to "Y", then control passes to the next line. If A$ is equal to "Y", then COUNT is incremented by 1, a message is displayed, and control is transferred to line 300.

```
>100 IF A$="Y" THEN COUNT=COU
 NT+1::DISPLAY AT(24,1):"HERE
 WE GO AGAIN!"::GOTO 300
```

IF HOURS<=40 THEN PAY=HOURS*WAGE ELSE PAY=HOURS*WAGE+.5*WAGE*(HOURS-40)::OT=1 operates as follows: If HOURS is less than or equal to 40, then PAY is set equal to HOURS*WAGE and control passes to the next line. If HOURS is greater than 40 then PAY is set equal to HOURS*WAGE+.5*WAGE*(HOURS-40), OT is set equal to 1, and control passes to the next line.

```
>100 IF HOURS<=40 THEN PAY=HO
 URS*WAGE ELSE PAY=HOURS*WAGE
 +.5*WAGE*(HOURS-40) :: OT=1
```

IF A=1 THEN IF B=2 THEN C=3 ELSE D=4 ELSE E=5 operates as follows: If A is not equal to 1, then E is set equal to 5 and control passes to the next line. If A is equal to 1 and B is not equal to 2, then D is set equal to 4 and control passes to the next line. If A is equal to 1 and B is equal to 2, then C is set equal to 3 and control passes to the next line.

```
>100 IF A=1 THEN IF B=2 THEN
 C=3 ELSE D=4 ELSE E=5
```

IF THEN [ELSE]

Program

The program on the right illustrates a use of IF-THEN-ELSE. It accepts up to 1000 numbers and then prints them in order from smallest to largest.

```
>100 CALL CLEAR
>110 DIM VALUE(1000)
>120 PRINT "ENTER VALUES TO B
 E SORTED.":"ENTER '9999' TO
 END ENTRY."
>130 FOR COUNT=1 TO 1000
>140 INPUT VALUE(COUNT)
>150 IF VALUE(COUNT)=9999 THE
 N 170
>160 NEXT COUNT
>170 COUNT=COUNT-1
>180 PRINT "SORTING."
>190 FOR SORT1=1 TO COUNT-1
>200 FOR SORT2=SORT1+1 TO COU
 NT
>210 IF VALUE(SORT1)>VALUE(SO
 RT2)THEN TEMP=VALUE(SORT1)::
  VALUE(SORT1)=VALUE(SORT2)::
  VALUE(SORT2)=TEMP
>220 NEXT SORT2
>230 NEXT SORT1
>240 FOR SORTED=1 TO COUNT
>250 PRINT VALUE(SORTED)
>260 NEXT SORTED
```


IMAGE

CHAPTER **4**

Format

IMAGE *format-string*

Description

The IMAGE statement specifies the format in which numbers are printed or displayed when the USING clause is present in PRINT or DISPLAY. No action is taken when the IMAGE statement is encountered during program execution. The IMAGE statement must be the only statement on a line. The following description of *format-string* also applies to the use of an explicit image after the USING clause in PRINT...USING and DISPLAY...USING.

Format-string must contain 254 or fewer characters and may be made up of any characters. They are treated as follows:

Pound signs (#) are replaced by the *print-list* values given in PRINT...USING or DISPLAY...USING. One pound sign must be allowed for each digit of the value and one for the negative sign if it is present, or for each character that is to be printed. If there is not enough room to print the number or characters in the space allowed, each pound sign is replaced with an asterisk (*). If more numbers are after the decimal place than are allowed by the number of pound signs after the decimal place in the IMAGE statement, the number is rounded to fit. If there are fewer non-numeric characters than are allowed for in the print string, the value printed will have blanks for the extra characters.

To indicate that a number is to be given in scientific notation, circumflexes (^) must be given for the E and power numbers. There must be four or five circumflexes, and 10 or fewer characters (minus sign, pound signs, and decimal point) when using the E format.

The decimal point separates the whole and fractional portions of numbers, and is printed where it appears in the IMAGE statement.

All other letters, numbers, and characters are printed exactly as they appear in the IMAGE statement.

Format-string may be enclosed in quotation marks. If it is not enclosed in quotation marks, leading and trailing spaces are ignored. However, when used directly in PRINT...USING or DISPLAY...USING, it must be enclosed in quotaton marks.

Each IMAGE statement may have space for many images, separated by any character except a decimal point. If more values are given in the PRINT...USING or DISPLAY...USING statement than there are images, then the images are reused, starting at the beginning of the statement.

If you wish, you may put *format-string* directly in the PRINT...USING or DISPLAY...USING statement immediately following USING. However, if a *format-string* is used often, it is more efficient to refer to an IMAGE statement.

Examples

IMAGE $####.### allows printing of any number from -999.999 to 9999.999. The following show how some sample values will be printed or displayed.

```
>100 IMAGE $####.###
>110 PRINT USING 100:A
```

Value	*Appearance*
-999.999	$-999.999
-34.5	$ -34.500
0	$ 0.000
12.4565	$ 12.457
6312.9991	$ 6312.999
99999999	$ ********

IMAGE THE ANSWERS ARE ### AND ##.## allows printing of two numbers. The first may be from -99 to 999 and the second may be from -9.99 to 99.99. The following show how some sample values will be printed or displayed.

```
>200 IMAGE THE ANSWERS ARE #
 ## AND ##.##
>210 PRINT USING 200:A,B
```

Values		*Appearance*
-99	-9.99	THE ANSWERS ARE -99 AND -9.99
-7	-3.459	THE ANSWERS ARE -7 AND -3.46
0	0	THE ANSWERS ARE 0 AND .00
14.8	12.75	THE ANSWERS ARE 15 AND 12.75
795	852	THE ANSWERS ARE 795 AND *****
-984	64.7	THE ANSWERS ARE *** AND 64.70

IMAGE DEAR ####, allows printing a four-character string. The following show how some sample values will be printed or displayed.

```
>300 IMAGE DEAR ####,
>310 PRINT USING 300:X$
```

Values	*Appearance*
JOHN	DEAR JOHN,
TOM	DEAR TOM ,
RALPH	DEAR ****,

Programs

The program on the right illustrates a use of IMAGE. It reads and prints seven numbers and their total. Lines 110 and 120 set up the images. They are the same except for the dollar sign in line 110. To keep the blank space where the dollar sign was, the *format-string* in line 120 is enclosed in quotation marks.

Line 180 prints the values using the IMAGE statements.

Line 210 shows that the format can be put directly in the PRINT...USING statement.

The amounts are printed with the decimal points lined up.

```
>100 CALL CLEAR
>110 IMAGE $####.##
>120 IMAGE " ####.##"
>130 DATA 233.45,-147.95,8.4,
 37.263,-51.299,85.2,464
>140 TOTAL=0
>150 FOR A=1 TO 7
>160 READ AMOUNT
>170 TOTAL=TOTAL+AMOUNT
>180 IF A=1 THEN PRINT USING
 110:AMOUNT ELSE PRINT USING
 120:AMOUNT
>190 NEXT A
>200 PRINT " -------"
>210 PRINT USING "$####.##":T
 OTAL
>RUN
$ 233.45
  -147.95
     8.40
    37.26
   -51.30
    85.20
   464.00
  -------
$ 629.06
```

The program at the right shows the effect of using more values in the PRINT...USING statement than there are images in the IMAGE statement.

```
>100 IMAGE ###.##,###.#
>110 PRINT USING 100:50.34,50
 .34,37.26,37.26
>RUN
 50.34, 50.3
 37.26, 37.3
```


INIT subprogram

Format

CALL INIT

Description

The INIT subprogram is used, along with LINK, LOAD, and PEEK, to access assembly language subprograms. The INIT subprogram checks to see that the Memory Expansion is connected, prepares the computer to run assembly language programs, and loads a set of supporting routines into the Memory Expansion.

The INIT subprogram must be called before LOAD and LINK are called. INIT removes any previously loaded subprograms from the Memory Expansion. The effects of INIT last until the Memory Expansion is turned off and does not need to be called from each program that is using the subprogram involved.

If the Memory Expansion is not attached, a syntax error is given.

INPUT

Format

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

(For information on using the INPUT statement with a file, see INPUT with files.)

Description

This form of the INPUT statement is used when entering data from the keyboard. The INPUT statement suspends program execution until data is entered from the keyboard. The optional *input-prompt* may display on the screen what data is expected.

Variable-list contains the variables (scalar or array elements: numeric or string) which are assigned values when the INPUT statement is executed. The variables are separated by commas. If a value in *variable-list* is input, it may later be used as a subscript in the same INPUT statement.

When inputting string values, they may optionally be enclosed in quotation marks. However, if you wish to have leading or trailing blanks or commas, the entire string *must* be enclosed in quotation marks. If more than one value is to be input, separate the values to be input by commas.

Options

The optional *input-prompt* is a string expression. It must be followed by a colon. It is displayed on the screen when the INPUT statement is executed. If there is no *input-prompt*, a question mark and space are displayed to indicate that input is expected. If there is an *input-prompt*, it takes the place of the question mark and space.

Examples

INPUT X allows the input of a number.

```
>100 INPUT X
```

INPUT X$,Y allows the input of a string and a number.

```
>100 INPUT X$,Y
```

INPUT "ENTER TWO NUMBERS: ":A,B prints the prompt ENTER TWO NUMBERS and then allows the entry of two numbers.

```
>100 INPUT "ENTER TWO NUMBERS
": A,B
```

INPUT A(J),J first evaluates the subscript of A and then accepts data into that subscript of A. Then a value is accepted into J.

```
>100 INPUT A(J),J
```


INPUT J,A(J) first accepts data into J and then accepts data into the Jth element of the array A.

```
>100 INPUT J,A(J)
```

Program

The program on the right illustrates a use of INPUT from the keyboard. Lines 110 through 140 allow the person using the program to enter data, as requested with the input-prompts.

Lines 170 through 250 construct a letter based on the input.

```
>100 CALL CLEAR
>110 INPUT "ENTER YOUR FIRST
 NAME: ":FNAME$
>120 INPUT "ENTER YOUR LAST N
 AME: ":LNAME$
>130 INPUT "ENTER A THREE DIG
 IT NUMBER: ":DOLLARS
>140 INPUT "ENTER A TWO DIGIT
  NUMBER: ":CENTS
>150 IMAGE OF $###.## AND THA
 T IF YOU
>160 CALL CLEAR
>170 PRINT "DEAR ";FNAME$;","
 : :
>180 PRINT "     THIS IS TO R
 EMIND YOU"
>190 PRINT "THAT YOU OWE US T
 HE AMOUNT"
>200 PRINT USING 150:DOLLARS+
 CENTS/100
>210 PRINT "DO NOT PAY US, YO
 U WILL SOON"
>220 PRINT "RECEIVE A LETTER
 FROM OUR"
>230 PRINT "ATTORNEY, ADDRESS
 ED TO"
>240 PRINT FNAME$;" ";LNAME$;
 "!": :
>250 PRINT TAB(15);"SINCERELY
 ,": : :TAB(15);"I. DUN YOU":
 : : :
>260 GOTO 260
 (Press SHIFT C to stop the
 program.)
```

INPUT (with files)

Format

INPUT *#file-number* [,REC *record-number*] :*variable-list*

(For information on using the INPUT statement to enter data from the keyboard, see INPUT.)

Description

The INPUT statement, when used with files, allows you to read data from files. The INPUT statement can only be used with files opened in INPUT or UPDATE mode. DISPLAY files may not have over 160 characters in each record.

File-number and *variable-list* must be included in the INPUT statement. *Record-number* may optionally be included when reading random access (RELATIVE) files from diskettes.

All statements which refer to files do so with a *file-number* from 0 through 255. *File-number* is assigned to a particular file by the OPEN statement. File number 0 is dedicated to the keyboard and screen of the computer. It cannot be used for other files and is always open. *File-number* is entered as a number sign (#) followed by a numeric expression that, when rounded to the nearest integer, is a number from 0 to 255, and is the number of a file that is open.

Variable-list is the list of variables into which you want the data from the file to be placed. It consists of string or numeric variables separated by commas with an optional trailing comma.

Options

You can optionally specify the number of the record that you want to read as *record-number*. It can only be specified for diskette files which have been opened as RELATIVE. The first record of a file is number 0.

Examples

INPUT #1:X$ puts into X$ the next value available in the file that was opened as #1.

```
>100 INPUT #1:X$
```

INPUT #23:X,A,LL$ puts into X, A, and LL$ the next three values from the file that was opened as #23.

```
>100 INPUT #23:X,A,LL$
```

INPUT #11,REC 44:TAX puts into TAX the first value of record number 44 of the file that was opened as #11.

```
>100 INPUT #11,REC 44:TAX
```

INPUT #3:A,B,C, puts into A, B, and C the next three values from the file that was opened as #3. The comma after C creates a pending input condition. When the next INPUT or LINPUT statement using this file is performed, one of the following actions occurs: If the next INPUT or LINPUT statement has no REC clause, the computer uses the data beginning where the previous INPUT statement stopped. If the next INPUT or LINPUT statement includes a REC clause, the computer terminates the pending input condition and reads the specified record.

```
>100 INPUT #3:A,B,C,
```

Program

The program at the right illustrates a use of the INPUT statement. It opens a file on the cassette recorder and writes 5 records on the file. It then goes back and reads the records and displays them on the screen.

```
>100 OPEN #1:"CS1",SEQUENTIAL
 ,INTERNAL,OUTPUT,FIXED 64
>110 FOR A=1 TO 5
>120 PRINT #1:"THIS IS RECORD
 ",A
>130 NEXT A
>140 CLOSE #1
>150 CALL CLEAR
>160 OPEN #1:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED 64
>170 FOR B=1 TO 5
>180 INPUT #1:A$,C
>190 DISPLAY AT(B,1):A$;C
>200 NEXT B
>210 CLOSE #1
>RUN
 * REWIND CASSETTE TAPE  CS1
 THEN PRESS ENTER
 * PRESS CASSETTE RECORD  CS1
 THEN PRESS ENTER
 * PRESS CASSETTE STOP    CS1
 THEN PRESS ENTER
 * REWIND CASSETTE TAPE   CS1
 THEN PRESS ENTER
 * PRESS CASSETTE PLAY    CS1
 THEN PRESS ENTER
 THIS IS RECORD 1
 THIS IS RECORD 2
 THIS IS RECORD 3
 THIS IS RECORD 4
 THIS IS RECORD 5
 * PRESS CASSETTE STOP    CS1
   THEN PRESS ENTER
```

See the Disk Memory System manual for instrucions on using diskettes.

INT

Format

INT(*numeric-expression*)

Description

The INT function returns the greatest integer less than or equal to *numeric-expression*.

Examples

PRINT INT(3.4) prints 3.

```
>100 PRINT INT(3.4)
```

X = INT(3.9) sets X equal to 3.

```
>100 X=INT(3.90)
```

P = INT(3.9999999999) sets P equal to 3.

```
>100 P=INT(3.9999999999)
```

DISPLAY AT(3,7):INT(4.0) displays 4 at the third row, seventh column.

```
>100 DISPLAY AT(3,7):INT(4.0)
```

N = INT(− 3.9) sets N equal to −4.

```
>100 N=INT(-3.9)
```

K = INT(− 3.0000001) sets K equal to −4.

```
>100 K=INT(-3.0000001)
```

JOYST subprogram

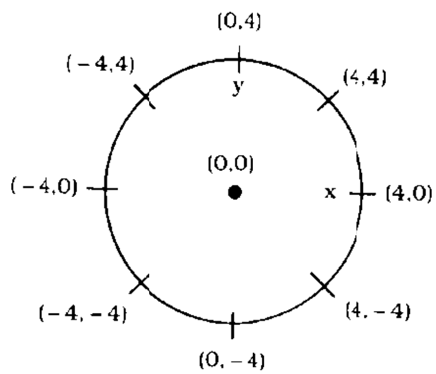
Format

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

Description

The JOYST subprogram returns data into *x-return* and *y-return* based on the position of the joystick in the Wired Remote Controller (available separately) labeled *key-unit*. *Key-unit* is a numeric expression with a value of 1 through 4. The values 1 and 2 are joysticks 1 and 2. Values 3 and 4 are reserved for possible future use.

The values returned in *x-return* and *y-return* depend on the position of the joystick. The values returned are shown below. The first value in the parentheses is placed in *x-return*. The second value is placed in *y-return*.



Example

CALL JOYST(1,X,Y) returns values in X and Y according to the position of joystick number 1.

```
>100 CALL JOYST(1,X,Y)
```

Program

The program on the right illustrates a use of the JOYST subprogram. It creates a sprite and then moves it around according to the input from a joystick.

```
>100 CALL CLEAR
>110 CALL SPRITE(#1,33,5,96,1
 28)
>120 CALL JOYST(1,X,Y)
>130 CALL MOTION(#1,-Y,X)
>140 GOTO 120
 (Press SHIFT C to stop the
 program.)
```


KEY subprogram

Format

CALL KEY(*key-unit,return-variable,status-variable*)

Description

The KEY subprogram assigns the code of the key pressed to *return-variable*. The value assigned depends on the *key-unit* specified. If *key-unit* is 0, input is taken from the entire keyboard, and the value placed in *return-variable* is the ASCII code of the key pressed. If no key is pressed, *return-variable* is set equal to -1. See *Appendix C* for a list of the ASCII codes.

If *key-unit* is 1, input is taken from the left side of the keyboard. If *key-unit* is 2, input is taken from the right side of the keyboard. The possible values placed in *return-variable* are given in *Appendix J*. Values of 3, 4, and 5 are reserved for possible future uses.

Status-variable indicates whether a key has been pressed. A value of 1 means a new key was pressed since the last CALL KEY was executed. A value of -1 means the same key was pressed as in the previous CALL KEY. A value of 0 means no key was pressed.

Example

CALL KEY(0,K,S) returns in K the ASCII code of any key pressed on the keyboard, and in S a value indicating whether any key was pressed.

```
>100 CALL KEY(0,K,S)
```

Program

The program on the right illustrates a use of the KEY subprogram. It creates a sprite and then moves it around according to the input from the left side of the keyboard.
Note that line 130 returns to line 120 if no key has been pressed.

```
>100 CALL CLEAR
>110 CALL SPRITE(#1,33,5,96,1
 28)
>120 CALL KEY(1,K,S)
>130 IF S=0 THEN 120
>140 IF K=5 THEN Y=-4
>150 IF K=0 THEN Y=4
>160 IF K=2 THEN X=-4
>170 IF K=3 THEN X=4
>180 IF K=1 THEN X,Y=0
>190 IF K>5 THEN X,Y=0
>200 CALL MOTION(#1,Y,X)
>210 GOTO 120
 (Press SHIFT C to stop the
 program.)
```

LEN

Format

LEN(*string-expression*)

Description

The LEN function returns the number of characters in *string-expression*. A space counts as a character.

Examples

PRINT LEN("ABCDE") prints 5.	`>100 PRINT LEN("ABCDE")`
X = LEN("THIS IS A SENTENCE.") sets X equal to 19.	`>100 X=LEN("THIS IS A SENTENC E.")`
DISPLAY LEN("") displays 0.	`>100 DISPLAY LEN("")`
DISPLAY LEN(" ") displays 1.	`>100 DISPLAY LEN(" ")`

LET

Format

[LET] *numeric-variable* [,*numeric-variable*, ...] = *numeric-expression*
[LET] *string-variable* [,*string-variable*, ...] = *string-expression*

Description

The LET statement assigns the value of an expression to the specified variable(s). The computer evaluates the expression on the right and puts its value into the variable(s) on the left. If more than one variable is on the left, they are separated with commas. The LET is optional, and is omitted in the examples in this manual. All subscripts in the variable(s) on the left are evaluated before any assignments are made.

You may use relational and logical operators in *numeric-expression*. If the relation or logical value is true, *numeric-variable* is assigned a value of −1. If the relation or logical value is false, *numeric-variable* is assigned a value of 0.

Examples

T = 4 puts the value 4 into T.	`>100 T=4`
X,Y,Z = 12.4 puts the value 12.4 into X, Y, and Z.	`>100 X,Y,Z=12.4`
A = 3 < 5 puts −1 into A since it is true that 3 is less than 5.	`>100 A=3<5`
B = 12 < 7 puts 0 into B since it is not true that 12 is less than 7.	`>100 B=12<7`
I,A(I) = 3 puts 3 into A(I) with whatever value I had before, and then puts 3 into I.	`>100 I,A(I)=3`
L$,D$,B$ = "B" puts "B" into L$, D$, and B$.	`>100 L$,D$,B$="B"`

LINK subprogram

Format

CALL LINK(*subprogram-name* [,*argument-list*])

Description

The LINK subprogram is used, along with INIT, LOAD, and PEEK, to access assembly language subprograms. The LINK subprogram passes control and, optionally, a list of parameters from a TI Extended BASIC program to an assembly language subprogram.

Subprogram-name is the name of the subprogram to be called. It must have been previously loaded into the Memory Expansion with the CALL LOAD command or statement. *Argument-list* is a list of variables and expressions as required by the specific assembly language subprogram being called.

LINPUT

Format

LINPUT [[#*file-number*] [,REC *record-number*] :] *string-variable*
LINPUT [*input-prompt*] *string-variable*

Description

The LINPUT statement allows the assignment of an entire line, file record, or (if there is a pending input record) the remaining portion of a file record into *string-variable*. No editing is performed on what is input, so commas, leading and trailing blanks, semicolons, colons, and quotation marks are placed in *string-variable* as they are given.

Options

A #*file-number* may be specified. If the file is in RELATIVE format, a specific record may be specified with REC. The file must be a DISPLAY-type file. If no file is specified, an *input-prompt* may be displayed prior to accepting input from the keyboard.

Examples

LINPUT L$ assigns into L$ anything typed before ENTER is pressed.

```
>100 LINPUT L$
```

LINPUT "NAME: ":NM$ displays NAME: and assigns into NM$ anything typed before ENTER is pressed.

```
>100 LINPUT "NAME: ":NM$
```

LINPUT #1,REC M:L$(M) assigns into L$(M) the value that was in record M of the file that was opened as #1.

```
>100 LINPUT #1,REC M:L$(M)
```

Program

The program on the right illustrates the use of LINPUT. It reads a previously existing file and displays only the lines that contain the word "THE".

```
>100 OPEN #1:"DSK1.TEXT1",INP
 UT,FIXED 80,DISPLAY
>110 IF EOF(1) THEN CLOSE #1
 :: STOP
>120 LINPUT #1:A$
>130 I=POS(A$,"THE",1)
>140 IF I<>0 THEN PRINT A$
>150 GOTO 110
```

LIST

Format

LIST ["*device-name*":] [*line-number*]
LIST ["*device-name*":] [*start-line-number*] – [*end-line-number*]

Description

The LIST command allows you to display program lines. If LIST is entered with no numbers following it, the entire program in memory is listed. If a number follows LIST, the line with that number is listed. If a number followed by a hyphen follows LIST, that line and all lines following it are listed. If a number preceeded by a hyphen follows LIST, all lines preceeding it and that line are listed. If two numbers separated by a hyphen follow LIST, the indicated lines and all lines between them are listed.

By pressing and holding a key until TI Extended BASIC responds, you may temporarily halt a listing so that you can look at it on the screen. Press any key again to restart the listing. Similarly, pressing **SHIFT C** (CLEAR) stops the listing.

Options

The listing normally is displayed on the screen. If you wish, you can instead direct the list to some other *device*, such as the optional thermal printer or RS232 interface, by specifying *device-name*.

Examples

LIST lists the entire program in memory on the display screen.	>LIST
LIST 100 lists line 100.	>LIST 100
LIST 100- lists line 100 and all lines after it.	>LIST 100-
LIST -200 lists all lines up to and including line 200.	>LIST -200
LIST 100-200 lists all lines from 100 through 200.	>LIST 100-200
LIST "TP" lists the entire program on the optional thermal printer.	>LIST "TP"
LIST "TP": -200 lists all lines up to and including line 200 on the optional thermal printer.	>LIST "TP": -200

LOAD subprogram

Format

CALL LOAD("*access-name*" [,*address,byte1* [, ...] ,*file-field*, ...])

Description

The LOAD subprogram is used, along with INIT, LINK, and PEEK, to access assembly language subprograms. The LOAD subprogram loads an assembly language object file or direct data into the Memory Expansion for later execution using the LINK statement.

The LOAD subprogram can specify one or more files from which to load object data or lists of direct load data, which consists of an *address* followed by data *bytes*. The *address* and data *bytes* are separated by commas. Direct load data must be separated by *file-field*, which is a string expression specifying a file from which to load assembly language object code. *File-field* may be a null string when it is used merely to separate direct load data fields. Use of the LOAD subprogram with incorrect values can cause the computer to cease to function and require turning it off and back on.

Assembly language subprogram names (see LINK) are included in the file.

LOCATE subprogram

Format

CALL LOCATE(*#sprite-number,dot-row,dot-column* [,...])

Description

The LOCATE subprogram is used to change the location of the given sprite(s) to the given *dot-row(s)* and *dot-column(s)*. *Dot-row* and *dot-column* are numbered consecutively starting with 1 in the upper left hand corner of the screen. *Dot-row* can be from 1 to 192 and *dot-column* can be from 1 to 256. (Actually *dot-row* can go up to 256, but the locations from 193 through 256 are off the bottom of the screen.) The location of the sprite is the upper left hand corner of the character(s) which define it.

Program

The program on the right illustrates the use of the LOCATE subprogram.
Line 110 creates a sprite as a fairly quickly moving red exclamation point.
Line 140 locates the sprite at a location randomly chosen in lines 120 and 130.
Line 150 repeats the process.

```
>100 CALL CLEAR
>110 CALL SPRITE(#1,33,7,1,1,
 25,25)
>120 YLOC=INT(RND*150+1)
>130 XLOC=INT(RND*200+1)
>140 CALL LOCATE(#1,YLOC,XLOC
 )
>150 GOTO 120
 (Press SHIFT C to stop the
 program.)
```

Also see the third example of the SPRITE subprogram.

LOG

Format

LOG(*numeric-expression*)

Description

The LOG function returns the natural logarithm of *numeric-expression* where *numeric-expression* is greater than zero. The LOG function is the inverse of the EXP function.

Examples

PRINT LOG(3.4) prints the natural logarithm of 3.4 which is 1.223775431622.

```
>100 PRINT LOG(3.4)
```

X = LOG(EXP(7.2)) sets X equal to the natural logarithm of e raised to the 7.2 power, which is 7.2.

```
>100 X=LOG(EXP(7.2))
```

S = LOG(SQR(T)) sets S equal to the natural logarithm of the square root of the value of T.

```
>100 S=LOG(SQR(T))
```

Program

The program at the right returns the logarithm of any positive number to any base.

```
>100 CALL CLEAR
>110 INPUT "BASE: ":B
>120 IF B<=1 THEN 110
>130 INPUT "NUMBER: ":N
>140 IF N<=0 THEN 130
>150 LG=LOG(N)/LOG(B)
>160 PRINT "LOG BASE";B;"OF";
 N;"IS";LG
>170 GOTO 110
 (Press SHIFT C to stop the
 program.)
```

MAGNIFY subprogram

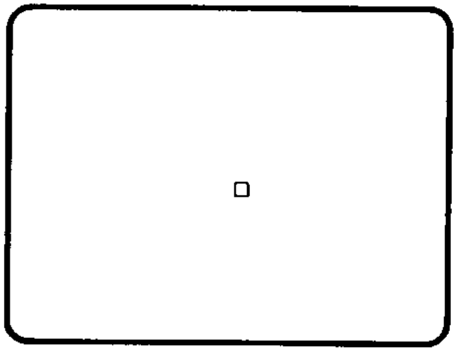
Format

CALL MAGNIFY(*magnification-factor*)

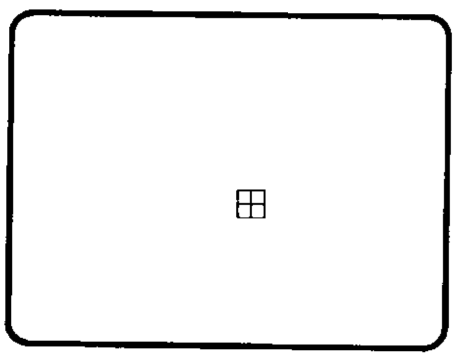
Description

The MAGNIFY subprogram allows you to specify the size of sprites and how many characters make up each sprite. All sprites are affected by MAGNIFY. *Magnification-factors* may be 1, 2, 3, or 4. If no CALL MAGNIFY is in a program, the default *magnification-factor* is 1.

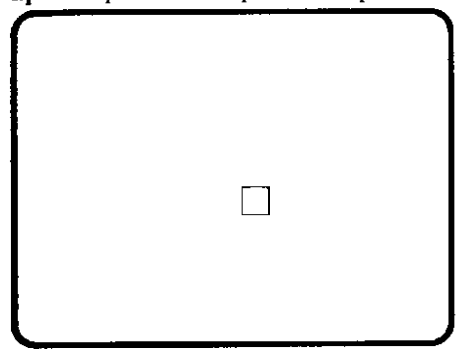
A *magnification-factor* of 1 causes all sprites to be single size and unmagnified. This means that each sprite is defined only by the character specified when the sprite was created and takes up just one character position on the screen.



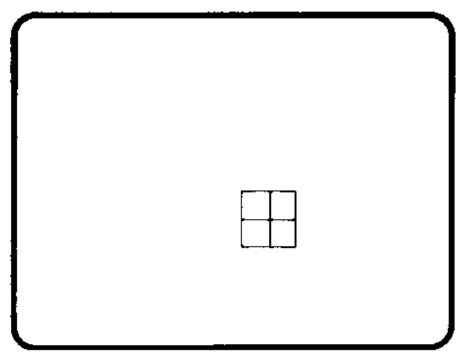
A *magnification-factor* of 2 causes all sprites to be single size and magnified. This means that each sprite is defined only by the character specified when it was created, but takes up four character positions on the screen. Each dot position in the character specified expands to occupy four dot positions on the screen. The expansion from a *magnification-factor* of 1 is down and to the right.



A *magnification-factor* of 3 causes all sprites to be double size and unmagnified. This means that each sprite is defined by four character positions that include the character specified. The first character is the one specified when the sprite was created if its number is evenly divisible by four, or the next smallest number that is evenly divisible by four. That character is the upper left quarter of the sprite. The next character is the lower left quarter of the sprite. The next character is the upper right quarter of the sprite. The final character is the lower right quarter of the sprite. The character specified when the sprite was created is one of the four that makes up the sprite. The sprite occupies four character positions on the screen.



A *magnification-factor* of 4 causes all sprites to be double size and magnified. This means that each sprite is defined by four character positions that include the character specified. The first character is the one specified when the sprite was created if its number is evenly divisible by four, or the next smallest number that is evenly divisible by four. That character is the upper left quarter of the sprite. The next character is the lower left quarter of the sprite. The next character is the upper right quarter of the sprite. The final character is the lower right quarter of the sprite. The character specified when the sprite was created is one of the four that makes up the sprite. The sprite occupies sixteen character positions on the screen. The expansion from a *magnification-factor* of 3 is down and to the right.



MAGNIFY subprogram

Program

The following program illustrates a use of the MAGNIFY subprogram. When it is run, a little figure appears near the center of the screen. In a moment, it gets to be twice as big, covering four character positions. In another moment, it is replaced by the upper left corner of a larger figure, still covering four character positions. Then the full figure appears, covering sixteen character positions. Finally it is reduced in size to four character positions.

Line 110 defines character 96.

Line 120 sets up a sprite using character 96. By default the magnification factor is 1.

Line 140 changes the magnification factor to 2.

Line 160 redefines character 96. Because the definition is 64 characters long, it also defines characters 97, 98, and 99.

Line 180 changes the magnification factor to 4.

Line 200 changes the magnification factor to 3.

```
>100 CALL CLEAR
>110 CALL CHAR(96,"1898FF3D3C
 3CE404")
>120 CALL SPRITE(#1,96,5,92,1
 24)
>130 GOSUB 230
>140 CALL MAGNIFY(2)
>150 GOSUB 230
>160 CALL CHAR(96,"0103C3417F
 3F07070707077E7C40000080C0C0
 80FCFEE2E3E0E0E06060606070")
>170 GOSUB 230
>180 CALL MAGNIFY(4)
>190 GOSUB 230
>200 CALL MAGNIFY(3)
>210 GOSUB 230
>220 STOP
>230 REM DELAY
>240 FOR DELAY=1 TO 500
>250 NEXT DELAY
>260 RETURN
```


MAX

Format

MAX(*numeric-expression1*,*numeric-expression2*)

Description

The MAX function returns the larger of *numeric-expression1* and *numeric-expression2*. If they are equal, then their value is returned.

Examples

PRINT MAX(3,8) prints 8.

```
>100 PRINT MAX(3,8)
```

F=MAX(3E12,1800000) sets F equal to 3E12.

```
>100 F=MAX(3E12,1800000)
```

G=MAX(−12,−4) sets G equal to −4.

```
>100 G=MAX(-12,-4)
```

L=MAX(A,B) sets L equal to 7 if A is 7 and B is −5.

```
>100 L=MAX(A,B)
```

MERGE

Format

MERGE ["] *device-filename* ["]

Description

The MERGE command merges lines in *filename* from the given *device* into the program lines already in the computer's memory. If a line number in *filename* duplicates a line number in the program already in memory, the new line replaces the old line. Otherwise the lines are inserted in line number order among the lines already in memory. The MERGE command does not clear breakpoints. Also, MERGE can *only* be used with diskettes.

NOTE: Files can only be merged into memory if they were saved using the MERGE option. See the SAVE command for more information.

Example

MERGE DSK1.SUB merges the program SUB into the program currently in memory.

```
>MERGE DSK1.SUB
```

Program

If the program on the right is saved on DSK1 as BOUNCE with the merge option, it can be merged with programs such as the one shown on the next page.

```
>100 CALL CLEAR
>110 RANDOMIZE
>140 DEF RND50=INT(RND*50-25)
>150 GOSUB 10000
>10000 FOR AA=1 TO 20
>10010 QQ=RND50
>10020 LL=RND50
>10030 CALL MOTION(#1,QQ,LL)
>10040 NEXT AA
>10050 RETURN

>SAVE "DSK1.BOUNCE",MERGE
```


On the right is a program you can put into the computer's memory.

```
>120 CALL CHAR(96,"18183CFFFF
 3C1818")
>130 CALL SPRITE(#1,96,7,92,1
 28)
>150 GOSUB 500
>160 STOP
```

Now merge BOUNCE with the above program.

```
>MERGE DSK1.BOUNCE
```

The program that results from merging BOUNCE with the above program is shown on the right.

Note that line 150 is from the program that was merged, not from the program that was in memory.

```
>LIST
>100 CALL CLEAR
>110 RANDOMIZE
>120 CALL CHAR(96,"18183CFFFF
 3C1818")
>130 CALL SPRITE(#1,96,7,92,1
 28)
>140 DEF RND50=INT(RND*50-25)
>150 GOSUB 10000
>160 STOP
>10000 FOR AA=1 TO 20
>10010 QQ=RND50
>10020 LL=RND50
>10030 CALL MOTION(#1,QQ,LL)
>10040 NEXT AA
>10050 RETURN
```

MIN

Format

MIN(*numeric-expression1*,*numeric-expression2*)

Description

The MIN function returns the smaller of *numeric-expression1* and *numeric-expression2*. If they are equal, then their value is returned.

Examples

PRINT MAX(3,8) prints 3.	>100 PRINT MAX(3,8)
F=MIN(3E12,1800000) sets F equal to 1800000.	>100 F=MIN(3E12,1800000)
G=MIN(-12,-4) sets G equal to -12.	>100 G=MIN(-12,-4)
L=MIN(A,B) sets L equal to -5 if A is 7 and B is -5.	>100 L=MIN(A,3)

MOTION subprogram

Format

CALL MOTION(*#sprite-number*,*row-velocity*,*column-velocity* [,...])

Description

The MOTION subprogram is used to specify the *row-velocity* and *column-velocity* of a sprite. If both the *row-* and *column-velocities* are zero, the sprite is stationary. A positive *row-velocity* moves the sprite down and a negative value moves it up. A positive *column-velocity* moves the sprite to the right and a negative value moves it to the left. If both *row-velocity* and *column-velocity* are nonzero, the sprite moves smoothly at an angle in a direction determined by the actual values.

The *row-* and *column-velocities* may be from -128 to 127. A value close to zero is very slow. A value far from zero is very fast. When a sprite comes to the edge of the screen, it disappears and reappears in the corresponding position on the other side of the screen.

Program

The program at the right illustrates a use of the MOTION subprogram.	>100 CALL CLEAR
Line 110 creates a sprite.	>110 CALL SPRITE(#1,33,5,92,1 24)
Lines 120 and 130 set values for the motion of the sprite.	>120 FOR XVEL=-16 TO 16 STEP 2 >130 FOR YVEL=-16 TO 16 STEP 2
Line 140 displays the current values of the motion of the sprite.	>140 DISPLAY AT (12,11):XVEL; YVEL
Line 150 sets the sprite in motion.	>150 CALL MOTION(#1,YVEL,XVEL)
Lines 160 and 170 complete the loops that set the values for the motion of the sprite.	>160 NEXT YVEL >170 NEXT XVEL

NEW

Format

NEW

Description

The NEW command clears the memory and screen and prepares the computer for a new program. All values are reset and all defined characters become undefined. Any open files are closed. Characters 32 through 95 are reset to their standard representations. The TRACE and BREAK commands are canceled.

Be sure to save the program that you have been working on before you enter NEW as it is unrecoverable by any means once NEW has been entered.

NEXT

Format

NEXT *control-variable*

See ON BREAK, ON WARNING, and RETURN (with ON ERROR) for the use of NEXT clause with those statements.

Description

The NEXT statement is always paired with the FOR-TO-STEP statement for construction of a loop. *Control-variable* must be the same as *control-variable* in the FOR-TO-STEP statement. The NEXT statement may not appear in an IF-THEN-ELSE statement.

The NEXT statement controls when the loop is repeated. Each time the NEXT statement is executed, *control-variable* is changed by the value following STEP in the FOR-TO-STEP statement, or by 1 if there is no STEP clause. If the value of *control-variable* is between initial-value and limit, the loop is executed again. If it is not, control passes to the statement after NEXT. Thus the value of *control-variable* at the end of the loop is always the first value outside the range of the FOR-TO-STEP statement. See FOR-TO-STEP for more information.

Program

The program on the right illustrates a use of the NEXT statement in lines 130 and 140.

```
>100 TOTAL=0
>110 FOR COUNT=10 TO 0 STEP -
 2
>120 TOTAL=TOTAL+COUNT
>130 NEXT COUNT
>140 FOR DELAY=1 TO 100::NEXT
  DELAY
>150 PRINT TOTAL,COUNT;DELAY
>RUN
  30            -2  101
```

NUMBER

Format

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

Description

The NUMBER command generates sequenced line numbers, allowing entry of program lines without typing the line numbers. If *initial-line* and *increment* are not specified, the line numbers start at 100 and increase in increments of 10. You may give the command at any time in the Command Mode. If a line already exists, the current line is displayed. You may type over it to replace it, alter it using the edit functions, or press **ENTER** to confirm it. To leave the NUMBER mode, press **ENTER** when a line comes up with no statements on it or press **SHIFT C** (CLEAR) when any line is displayed. NUMBER may be abbreviated as NUM.

Options

You may specify an *initial-line* and/or *increment*.

Example

In the following, what you type is UNDERLINED. Press **ENTER** after each line.

NUM instructs the computer to number starting at 100 with increments of 10.

```
>NUM
>100 X=4
>110 Z=10
>120
```

NUM 110 instructs the computer to number starting at 110 with increments of 10. Change line 110 to Z = 11.

```
>NUM 110
 110 Z=11
>120 PRINT (Y+X)/Z
>130
```

NUM 105,5 instructs the computer to number starting at line 105 with increments of 5.
Line 110 already exists.

```
>NUM 105,5
>105 Y=7
 110 Z=11
>115
>LIST
 100 X=4
 105 Y=7
 110 Z=11
 120 PRINT (Y+X)/Z
```


OLD

Format

OLD ["] *device-program-name* ["]

Description

The OLD command loads *program-name* from *device* into memory. The program must first have been put on *device* using the SAVE command. OLD closes any open files and removes the program currently in memory before loading *program-name*. To add program lines from another program to a program in memory, see the MERGE command.

Device can be several different things. If it is CS1 or CS2, designating one of the two possible cassette recorders, then no *program-name* is given. The program loaded is the program that is on the cassette. Instructions on operating the cassette recorder are displayed on the screen.

See the Disk Memory System Manual for instructions on using OLD with diskettes.

Examples

OLD CS1 loads a program from a cassette recorder into the computer's memory.

```
>OLD CS1
```

OLD"DSK1.MYPROG" loads the program MYPROG into the computer's memory from the diskette in disk drive one.

```
>OLD "DSK1.MYPROG"
```

OLD DSK.DISK3.UPDATE80 loads the program UPDATE80 into the computer's memory from the diskette named DISK3.

```
>OLD DSK.DISK3.UPDATE80
```

ON BREAK

Format
ON BREAK STOP
ON BREAK NEXT

Description

The ON BREAK statement determines the action taken if a breakpoint is encountered during the execution of a program. The default action is STOP, which causes program execution to halt and the standard breakpoint message to be printed. The alternative is NEXT, which transfers control to the next line without a breakpoint occurring.

You can use ON BREAK NEXT to have a program ignore breakpoints which you have put in a program for debugging purposes. (NOTE: ON BREAK NEXT does not have any effect on a BREAK statement which is not followed by a program line number. The breakpoint will occur even if the statement ON BREAK NEXT has been executed.) When ON BREAK NEXT is in effect, the external break, **SHIFT C** (CLEAR), does not stop a program. In that case only **SHIFT Q** (QUIT) can stop the program. **SHIFT Q** (QUIT) erases the program and returns you to the main screen and may interfere with the proper operation of some external devices such as disk drives.

Program

The program on the right illustrates the use of ON BREAK. Line 110 sets a breakpoint in line 150. Line 120 sets breakpoint handling to go to the next line. A breakpoint occurs in line 130 in spite of line 120. Enter CONTINUE. No breakpoint occurs in line 150 because of line 120. **SHIFT C** (CLEAR) has no effect during the execution of lines 140 through 160 because of line 120. Line 170 restores the normal use of **SHIFT C** (CLEAR).

```
>100 CALL CLEAR
>110 BREAK 150
>120 ON BREAK NEXT
>130 BREAK
>140 FOR A=1 TO 50
>150 PRINT "SHIFT C IS DISABL
 ED."
>160 NEXT A
>170 ON BREAK STOP
>180 FOR A=1 TO 50
>190 PRINT "NOW IT WORKS."
>200 NEXT A
```


ON ERROR

Format
ON ERROR STOP
ON ERROR *line-number*

Description

The ON ERROR statement determines the action taken if an error occurs during the execution of a program. The default action is STOP, which causes the standard error message to be printed and program execution to halt. The alternative is to give a *line-number* which transfers control to that line in case of an error.

Once an error has occurred and control has been transferred, error handling reverts to the normal action, STOP. If you wish to have any new errors handled differently, an ON ERROR statement must be executed again.

If a *line-number* is specified by ON ERROR, the *line-number* must be the beginning of a subroutine similar to that called by GOSUB. It should end with a RETURN statement. See RETURN (with ON ERROR) for more information.

NOTE: A transfer of control following the execution of an ON ERROR statement acts like the execution of a GOSUB statement. As with GOTO and GOSUB, you must avoid transfers to and from subprograms. The most common result of an illegal transfer into a subprogram is a syntax error on a statement that appears to be correct.

Program

The program at the right illustrates the use of ON ERROR. Line 110 causes any error to pass control to line 160.

An error occurs in line 130 and control is passed to line 160.

Line 170 causes the *next* error to pass control to line 230. Line 180 finds out about the error using CALL ERR.

Line 190 transfers control to line 230 if the error isn't in the expected line.

Line 200 transfers control to line 230 if the error isn't the one expected.

Line 210 changes the value of XS to an acceptable value. Line 220 returns control to the line in which the error occurred.

Line 240 reports the nature of the unexpected error and the program stops.

```
>100 CALL CLEAR
>110 ON ERROR 160
>120 X$="A"
>130 X=VAL(X$)
>140 PRINT X;"SQUARED IS";X*X
>150 STOP
>160 REM ERROR SUBROUTINE
>170 ON ERROR 230
>180 CALL ERR(CODE,TYPE,SEVER
 ,LINE)
>190 IF LINE<>130 THEN RETURN
 230
>200 IF CODE<>74 THEN RETURN
 230
>210 X$="5"
>220 RETURN
>230 REM UNKNOWN ERROR
>240 PRINT "ERROR";CODE;" IN
LINE";LINE
>RUN
  5 SQUARED IS 25
```


ON GOSUB

CHAPTER 4

Format

ON *numeric-expression* GOSUB *line-number* [,...]
ON *numeric-expression* GO SUB *line-number* [,...]

Description

The ON...GOSUB statement transfers control to the subroutine beginning at *line-number* in the position corresponding to the value of *numeric-expression*. Other than giving a choice, it acts the same as the GOSUB statement, but it is more efficient in that it may require fewer lines of code than using an IF-THEN-ELSE statement.

Numeric-expression must have a value from 1 through the number of *line-numbers*.

Examples

ON X GOSUB 1000,2000,300 transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3.

```
>100 ON X GOSUB 1000,2000,300
```

ON P-4 GOSUB 200,250,300, 800,170 transfers control to 200 if P-4 is 1 (P is 5), 250 if P-4 is 2, 300 if P-4 is 3, 800 if P-4 is 4, and 170 if P-4 is 5.

```
>100 ON P-4 GOSUB 200,250,300
 ,800,170
```

Program

The program on the right illustrates a use of ON...GOSUB. Line 220 determines where to go according to the value of CHOICE.

```
>100 CALL CLEAR
>110 DISPLAY AT(11,1):"CHOOSE
 ONE OF THE FOLLOWING:"
>120 DISPLAY AT(13,1):"1 ADD
 TWO NUMBERS."
>130 DISPLAY AT(14,1):"2 MUL
TIPLY TWO NUMBERS."
>140 DISPLAY AT(15,1):"3 SUB
TRACT TWO NUMBERS."
>150 DISPLAY AT(20,1):"YOUR C
HOICE:"
>160 DISPLAY AT(22,2):"FIRST
NUMBER:"
>170 DISPLAY AT(23,1):"SECOND
 NUMBER:"
>180 ACCEPT AT (20,14)VALIDAT
E (NUMERIC):CHOICE
>190 IF CHOICE<1 OR CHOICE>3
THEN 180
>200 ACCEPT AT (22,16)VALIDAT
E(NUMERIC):FIRST
>210 ACCEPT AT (23,16)VALIDAT
E(NUMERIC):SECOND
>220 ON CHOICE GOSUB 240,260,
280
>230 GOTO 180
>240 DISPLAY AT(3,1):FIRST;"P
LUS";SECOND;"EQUALS";FIRST+S
ECOND
>250 RETURN
>260 DISPLAY AT(3,1):FIRST;"T
IMES";SECOND;"EQUALS";FIRST*
SECOND
>270 RETURN
>280 DISPLAY AT(3,1):FIRST;"M
INUS";SECOND;"EQUALS";FIRST-
SECOND
>290 RETURN
(Press SHIFT C to stop the
program.)
```


ON GOTO

Format

ON *numeric-expression* GOTO *line-number* [,...]
ON *numeric-expression* GO TO *line-number* [,...]

Description

The ON...GOTO statement transfers control to the *line-number* in the position corresponding to the value of *numeric-expression*. Other than giving a choice, it acts the same as the GOTO statement, but it is more efficient in that it may require fewer lines of code than using an IF-THEN-ELSE statement.

Numeric-expression must have a value from 1 through the number of *line-numbers*.

Examples

ON X GOTO 1000,2000,300 transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3. The equivalent statement using an IF-THEN-ELSE statement is IF X=1 THEN 1000 ELSE IF X=2 THEN 2000 ELSE IF X=3 THEN 300 ELSE PRINT "ERROR!"::STOP.

```
>100 ON X GOTO 1000,2000,300
```

ON P-4 GOTO 200,250,300,800,170 transfers control to 200 if P-4 is 1 (P is 5), 250 if P-4 is 2, 300 if P-4 is 3, 800 if P-4 is 4, and 170 if P-4 is 5.

```
>100 ON P-4 GOTO 200,250,300,
800,170
```

Program

The program on the right illustrates a use of ON...GOTO. Line 220 determines where to go according to the value of CHOICE.

```
>100 CALL CLEAR
>110 DISPLAY AT(11,1):"CHOOSE
 ONE OF THE FOLLOWING:"
>120 DISPLAY AT(13,1):"1 ADD
 TWO NUMBERS."
>130 DISPLAY AT(14,1):"2 MUL
TIPLY TWO NUMBERS."
>140 DISPLAY AT(15,1):"3 SUB
TRACT TWO NUMBERS."
>150 DISPLAY AT(20,1):"YOUR C
HOICE:"
>160 DISPLAY AT(22,2):"FIRST
NUMBER:"
>170 DISPLAY AT(23,1):"SECOND
 NUMBER:"
>180 ACCEPT AT (20,14)VALIDAT
E (NUMERIC):CHOICE
>190 IF CHOICE<1 OR CHOICE>3
THEN 180
>200 ACCEPT AT (22,16)VALIDAT
E NUMERIC):FIRST
>210 ACCEPT AT (23,16)VALIDAT
E NUMERIC):SECOND
>220 ON CHOICE GOTO 230,250,2
70
>230 DISPLAY AT(3,1):FIRST;"P
LUS";SECOND;"EQUALS";FIRST+S
ECOND
>240 GOTO 180
>250 DISPLAY AT(3,1):FIRST;"T
IMES";SECOND;"EQUALS";FIRST*
SECOND
>260 GOTO 180
>270 DISPLAY AT(3,1):FIRST;"M
INUS";SECOND;"EQUALS";FIRST-
SECOND
>280 GOTO 180
(Press SHIFT C to stop the
program.)
```


ON WARNING

Format

ON WARNING PRINT
ON WARNING STOP
ON WARNING NEXT

Description

The ON WARNING statement determines the action taken if a warning occurs during the execution of a program. The default action is PRINT, which causes the standard warning message to be printed and the program to continue execution. One alternative is STOP, which causes the standard warning message to be printed and the program to halt execution. The other alternative is NEXT which causes the program to continue execution without printing any message.

Program

The program on the right illustrates the use of ON WARNING. Line 110 sets warning handling to go to the next line. Line 120 therefore prints the result without any message.

```
>100 CALL CLEAR
>110 ON WARNING NEXT
>120 PRINT 120,5/0
```

Line 130 sets warning handling to the default, printing the message and then continuing execution. Line 140 therefore prints 140, then the warning, and then continues.

```
>130 ON WARNING PRINT
>140 PRINT 140,5/0
```

Line 150 sets warning handling to print the warning message and then stop execution. Line 160 therefore prints 160 and then the warning message and then stops.

```
>150 ON WARNING STOP
>160 PRINT 160,5/0
>170 PRINT 170
>RUN
  120             9.99999E+**
  140
 * WARNING
 NUMERIC OVERFLOW IN 140
                  9.99999E+**
  160
 * WARNING
 NUMERIC OVERFLOW IN 160
```

OPEN

Format

OPEN #*file-number*:*device-filename* [,*file-organization*] [,*file-type*] [,*open-mode*] [,*record-type*]

Description

The OPEN statement prepares a BASIC program to use data files stored on a diskette or cassette by providing a link between *file-number* and a file. To set up this link, the OPEN statement describes a file's characteristics. If the file already exists, the description that is given in the program must match the actual characteristics of the file. Files on cassettes are not checked, however, so errors may occur if the characteristics do not match.

File-number must be included in the OPEN statement. Statements which refer to files do so with a *file-number* from 0 through 255. File number 0 is the keyboard and screen of the computer. It cannot be used for other files and is always open. You may assign the other numbers as you wish, with each file having a different number. *File-number* is entered as a number sign (#) followed by a numeric expression that, when rounded to the nearest integer, is a number from 0 to 255, and is not the number of a file that is already open.

Device must also be included in the OPEN statement. If *device* is CS1 or CS2, designating one of the two cassette recorders, then no *file-name* is given. Instructions on operating the cassette recorder are displayed on the screen.

If *device* is DSK1, DSK2, or DSK3, designating one of the three disk drives, then *file-name* is the name of a file on the diskette in the given drive. If *device* is DSK.*diskette-name*, where *diskette-name* is the name of a diskette in one of the drives, then *file-name* is the name of a file on the diskette named *diskette-name*. The computer searches the drives, starting at DSK1, until it finds the diskette with the given name. Then it looks for *file-name* on

The other information may be in any order, or may be omitted. If an item is omitted, the computer assumes certain defaults, which are described below.

File-organization can be either sequential or random. Records in a sequential file are read or written one after the other. Records in random files can be read or written in any order. Random files may also be processed sequentially. To indicate which structure the file has, enter either SEQUENTIAL for sequential files or RELATIVE for random files. 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 do not specify the *file-organization*, the default is SEQUENTIAL.

File-type may be either DISPLAY or INTERNAL. Files can be written either in human-readable form, called ASCII (DISPLAY), or in machine-readable form, called binary (INTERNAL). Binary records may take up less space and are processed more quickly by the computer. However, if the information is going to be printed or displayed, ASCII format is usually a better choice.

To specify that you wish the file to be in ASCII format, enter DISPLAY. To specify binary format, enter INTERNAL. If you do not specify a *file-type*, the default is DISPLAY. Usually INTERNAL is the best choice when using files on cassettes or diskettes, and DISPLAY is the best choice when using files on the thermal printer or RS232 Interface.

Open-mode may be UPDATE, INPUT, OUTPUT, or APPEND. The computer may be instructed that the file may be both read and written on, that it may only be read, that it may only be written on, or that it may only be added to. However, if the file is marked as protected, it cannot be written on and may only be opened for input.

To be able both to read from and write to a file, specify UPDATE. To just read from a file, specify INPUT. To just write to a file, specify OUTPUT. To only add to a file, specify APPEND. Append mode can only be specified for VARIABLE length records. If you do not specify an *open-mode*, the default is UPDATE.

Note that if an unprotected file already exists on a diskette, specifying an *open-mode* of OUTPUT to the same file name writes over the existing file with the new data. You can prevent this by moving to the end of the file by using the RESTORE statement with the proper record or opening the file in the APPEND mode.

Record-type may be either VARIABLE or FIXED. Files may have records that are all the same length or that vary in length. If all records are the same length, any that are shorter are padded to make up the difference. Any that are longer may be truncated to the proper length. You may specify records of variable length by entering VARIABLE. You specify records of fixed length by entering FIXED.

If you like, you may specify a maximum length of a record by following VARIABLE or FIXED with a numeric expression. The maximum record is dependent on the device used. If you do not specify a record length, the default is 80 for diskettes, 64 for cassettes, 80 for the RS232 interface, and 32 for the thermal printer.

RELATIVE files must have FIXED length records. If you do not specify a *record-type* for a RELATIVE file, the default is FIXED.

SEQUENTIAL files may be either FIXED or VARIABLE. If you do not specify a *record-type* for a SEQUENTIAL file, the default is VARIABLE. A fixed-length file may be reopened for either SEQUENTIAL or RELATIVE access independent of previous *file-organization* assignments.

Examples

OPEN #1:"CS1",FIXED,OUTPUT opens a file on cassette one. The file is SEQUENTIAL, kept in DISPLAY format, in OUTPUT mode with FIXED length records with a maximum length of 64 bytes.

```
>100 OPEN #1:"CS1",FIXED,OUT
 PUT
```

OPEN #23:"DSK.MYDISK.X", RELATIVE 100,INTERNAL,UPDATE, FIXED opens a file named "X". The file is on the diskette named MYDISK in whichever drive that diskette it is located. The file is RELATIVE, kept in INTERNAL format with FIXED length records with a maximum Length of 80 bytes. The file is opened in UPDATE mode and starts with 100 records made available for it.

```
>300 OPEN #23:"DSK.MYDISK.X",
 RELATIVE 100,INTERNAL,UPDATE
 ,FIXED
```

OPEN #243:A$,INTERNAL, if A$ equals "DSK2.ABC", assumes a file on the diskette in drive two with a name of ABC. The file is

format, in UPDATE mode with VARIABLE length records with a maximum length of 80 bytes.

```
>100 OPEN #243:A$,INTERNAL
```

OPEN #17:"TP",OUTPUT prepares the thermal printer for printing.

```
>100 OPEN #17:"TP",OUTPUT
```


OPTION BASE

Format

OPTION BASE 0
OPTION BASE 1

Description

The OPTION BASE statement sets the lowest allowable subscript of arrays to zero or one. The default is zero. If an OPTION BASE statement is used, it must have a lower line number than any DIM statement or reference to any array. There may only be one OPTION BASE statement in a program, and it applies to all array subscripts. The OPTION BASE statement may not appear in an IF-THEN-ELSE statement.

Example

OPTION BASE 1 sets the lowest allowable subscript of all arrays to one.

```
>100 OPTION BASE 1
```

PATTERN subprogram

Format

CALL PATTERN(*#sprite-number,character-value* [,...])

Description

The PATTERN subprogram allows you to change the character pattern of a sprite without affecting any other characteristics of the sprite.

Sprite-number specifies the sprite you are using. *Character-value* may be any integer from 32 to 143. See the CHAR subprogram for information on defining the pattern for characters. See the MAGNIFY subprogram for more information.

Program

The program on the right illustrates the use of the PATTERN subprogram. Lines 110 through 140 build a floor.

```
>100 CALL CLEAR
>110 CALL COLOR(12,16,16)
>120 FOR A=19 TO 24
>130 CALL HCHAR(A,1,120,32)
>140 NEXT A
```

Lines 150 though 200 define characters 96 through 107.

```
>150 A$="01071821214141FFFF41
 41212119070080E09884848282FF
 FF8282848498E000"
>160 B$="01061820305C46818142
 46242C1807008060183424624281
 81623A0C0418E000"
>170 C$="0106182C244642818146
 5C3020180700806018040C3A6281
 814262243418E000"
>180 CALL CHAR(96,A$)
>190 CALL CHAR(100,B$)
>200 CALL CHAR(104,C$)
```

Line 210 creates a sprite in the shape of a wheel and starts it moving

```
>210 CALL SPRITE(#1,96,5,130,
 1,0,8)
```

Line 220 makes the sprite double size.

```
>220 CALL MAGNIFY(3)
```

Lines 230 through 270 make the spokes of the wheel appear to move as the character displayed is changed.

```
>230 FOR A=96 TO 104 STEP 4
>240 CALL PATTERN(#1,A)
>250 FOR DELAY=1 TO 5:: NEXT
 DELAY
>260 NEXT A
```

Also see the third example of the SPRITE subprogram.

```
>270 GOTO 230
 (Press SHIFT C to stop the
 program.)
```


PEEK subprogram

Format

CALL PEEK(*address,numeric-variable-list*)

Description

The PEEK subprogram is used, along with INIT, LINK, and LOAD, to access assembly language subprograms. The PEEK subprogram returns values in the variables in *numeric-variable-list* that correspond with the values in the byte specified by *address* and the bytes following it. PEEK can be used without assembly language subprograms, but the information obtained is of little use.

Detailed instructions on the use of INIT, LINK, LOAD, and PEEK are included with custom written programs that may be available on diskette or cassette.

Indiscriminate use of PEEK may cause the computer to "lock up" and require it to be turned off and back on before further use.

Example

CALL PEEK(8192,X1,X2,X3,X4) returns the values in locations 8192, 8193, 8194, and 8195 in X1, X2, X3, and X4, respectively.

```
>100 CALL PEEK(8192,X1,X2,X3,
 X4)
```

PI

Format

PI

Description

The PI function returns the value of π as 3.14159265359.

Example

VOLUME=4/3*PI*6∧3 sets VOLUME equal to four thirds times pi times six cubed, which is the volume of a sphere with a radius of six.

```
>100 VOLUME=4/3*PI*6∧3
```


POS

Format

POS(*string1,string2,numeric-expression*)

Description

The POS function returns the position of the first occurance of *string2* in *string1*. The search begins at the position specified by *numeric-expression*. If no match is found, the function returns a value of zero.

Examples

X=POS("PAN","A",1) sets X equal to 2 because A is the second letter in PAN.

```
>100 X=POS("PAN","A",1)
```

Y=POS("APAN","A",2) sets Y equal to 3 because the A in the third position in APAN is the first occurance of A in the portion of APAN that was searched.

```
>100 Y=POS("APAN","A",2)
```

Z=POS("PAN","A",3) sets Z equal to 0 because A was not in the part of PAN that was searched.

```
>100 Z=POS("PAN","A",3)
```

R=POS("PABNAN","AN",1) sets R equal to 5 because the first occurance of AN starts with the A in the fifth position in PABNAN.

```
>100 R=POS("PABNAN","AN",1)
```

Program

The program at the right illustrates a use of POS. In it any input is searched for spaces, and is then printed with each word on a single line.

```
>100 CALL CLEAR
>110 PRINT "ENTER A SENTENCE.
 "
>120 INPUT X$
>130 S=POS(X$," ",1)
>140 IF S=0 THEN PRINT X$ ::
 PRINT :: GOTO 110
>150 Y$=SEG$(X$,1,S):: PRINT
  Y$
>160 X$=SEG$(X$,S+1,LEN(X$))
>170 GOTO 130
 (Press SHIFT C to stop the
 program.)
```

POSITION subprogram

Format

CALL POSITION(*#sprite-number,dot-row,dot-column* [,...])

Description

The POSITION subprogram returns the position of the specified sprite(s) in the given *dot-row*(s) and *dot-column*(s) as numbers from 1 to 256. They are the position of the upper left corner of the sprite. If the sprite is not defined, *dot-row* and *dot-column* are set to zero.

The sprite continues to move after its position is returned, so that must be allowed for. The distance moved depends on the sprite's speed.

Example

CALL POSITION(#1,Y,X) returns the position of the upper left hand corner of sprite #1.

```
>100 CALL POSITION(#1,Y,X)
```

Also see the third example of the SPRITE subprogram.

CHAPTER 4

INT

at

T [*#file-number* [,REC *record-number*] :] [*print-list*]

iption

PRINT statement allows you to transfer the values of the elements of the nal *print-list* to the display screen or optionally to an external file or ce. *Print-list* consists of string constants, numeric constants, string bles, numeric variables, numeric expressions, string expressions, and/or TAB function. Each element in *print-list* is separated from the others by micolon, a comma, or a colon.

semicolon, comma, and colon control spacing for the screen or a file ned in DISPLAY format. A semicolon causes the next element to be ced immediately adjacent to the previous element. A comma causes the t element of *print-list* to be put in the next print field. Each print field is characters long. The number of print fields depends on the record length he device being used. On the screen, the print fields are at positions 1 and If the cursor is past the start of the last print field, the next item is nted on the next line. A colon causes the next element to be put on the xt line or record. To print several blank lines, you may put several colons er the PRINT statement. However, they must have spaces between them they are not confused with the statement separator symbol (::).

separator may be placed following the last element of *print-list*, which ects the placement of the next element of the next PRINT, PRINT...USING, SPLAY (without AT), or DISPLAY...USING (without AT) statement written the same device. It causes the next output statement to be considered to a continuation of the current one unless it is a PRINT statement with a EC clause.

hen printing a new line on the screen, everything (except sprites) is rolled up one line (so the top line is lost) and the new line is printed at the ottom of the screen.

ptions

The *#file-number* determines the file that is to be printed on. If it is omitted or #0, the screen is used. Otherwise *file-number* must be the number of a file that is already open. See OPEN.

The REC clause is used to specify the record on which you wish to print the elements in *print-list*. REC may only be used with files that were opened as RELATIVE files. See OPEN.

In printing to INTERNAL format files, the comma and semicolon both place the elements in *print-list* adjacent to each other. In DISPLAY format files, the comma and semicolon act as described above, with the semicolon placing the element adjacent to the previous element and the comma putting the element in the next print field.

Examples

PRINT causes a blank line to appear on the display screen.

```
>100 PRINT
```

PRINT "THE ANSWER IS":ANSWER causes the string constant THE ANSWER IS to be printed on the display screen, followed immediately by the value of ANSWER. If ANSWER is positive, there will be a blank for the positive sign after IS.

```
>100 PRINT "THE ANSWER IS";AN
 SWER
```

PRINT X:Y/2 causes the value of X to be printed on a line and the value of Y/2 to be printed on the next line.

```
>100 PRINT X:Y/2
```

PRINT #12,REC 7:A causes the value of A to be printed on the eighth record of the file that was opened as number 12. (Record number 0 is the first record.)

```
>100 PRINT #12,REC 7:A
```

PRINT #32 A,B,C, causes the values of A, B, and C to be printed on the next record of the file that was opened as number 32. The final comma creates a pending output condition. The next PRINT statement directed to file number 32 will print on the same record as this PRINT statement unless it specifies a record, thereby closing the pending output condition.

```
>100 PRINT #32:A,B,C,
```


PRINT #1,REC 3:A,B followed by PRINT #1:C,D causes A and B to be printed in record 3 of the file that was opened as number 1 and C and D to be printed in record 4 of the same file.

```
>100 PRINT #1,REC 3:A,B
>150 PRINT #1:C,D
```

Program

The program at the right prints out various values in various positions on the display screen.

```
>100 CALL CLEAR
>110 PRINT 1;2;3;4;5;6;7;8;9
>120 PRINT 1,2,3,4,5,6
>130 PRINT 1:2:3
>140 PRINT
>150 PRINT 1;2;3;
>160 PRINT 4;5;6/4
>RUN
 1  2  3  4  5  6  7  8  9
 1             2
 3             4
 5             6
 1
 2
 3

 1  2  3  4  5  1.5
```

PRINT USING

Format

PRINT [#*file-number* [,REC *record-number*]]USING *string-expression:print-list*
PRINT [#*file-number* [,REC *record-number*]]USING *line-number:print-list*

Description

The PRINT...USING statement acts the same as PRINT with the addition of the USING clause, which specifies the format to be used. *String-expression* defines the format in the manner described in IMAGE. *Line-number* refers to the line number of an IMAGE statement. See the IMAGE statement for more information on the use of *string-expression*.

Examples

PRINT USING "###.##":32.5 prints 32.50.

```
>100 PRINT USING "###.##":32.
 5
```

PRINT USING "THE ANSWER IS ###.#":123.98 prints THE ANSWER IS 124.0.

```
>100 PRINT USING "THE ANSWER
 IS ###.#":123.98
```

PRINT USING 185:37.4,-86.2 prints the values of 37.4 and -86.2 using the IMAGE statement in line 185.

```
>100 PRINT USING 185:37.4,-86
 .2
```


RANDOMIZE

Format

RANDOMIZE [*numeric-expression*]

Description

The RANDOMIZE statement resets the random number generator to an unpredictable sequence. If RANDOMIZE is followed by a *numeric-expression*, the same sequence of random numbers is produced each time the statement is executed with that value for the expression. Different values give different sequences.

Program

The program at the right illustrates a use of the RANDOMIZE statement. It accepts a value for numeric-expression and prints the first 10 values obtained using the RND function.

```
>100 CALL CLEAR
>110 INPUT "SEED: ":S
>120 RANDOMIZE S
>130 FOR A=1 TO 10::PRINT A;R
 ND::NEXT A::PRINT
>140 GOTO 110
 (Press SHIFT C to stop the
 program.)
```

REM

Format

REM *character-string*

Description

The REM statement allows you to enter remarks into your program. The remarks may be anything that you wish, but are usually used to divide sections of programs and to explain what the following section is meant to do. No matter what follows REM, including the statement separator symbol (::), remarks are not executed and have no effect on program execution. They do, however, take up space in memory.

Example

REM BEGIN SUBROUTINE identifies a section beginning a subroutine.	`>100 REM BEGIN SUBROUTINE`

RESEQUENCE

Format

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

Description

The RESEQUENCE command changes the line numbers of the program in memory. If no *initial-line* is given, the line numbering starts with 100. If no *increment* is given, an *increment* of 10 is used. RESEQUENCE may be abbreviated as RES.

In addition to renumbering lines, any line references in the statements BREAK, DISPLAY...USING, GOSUB, GOTO, IF-THEN-ELSE, ON ERROR, ON...GOSUB, ON...GOTO, PRINT...USING, RESTORE, RETURN, and RUN are also changed so that they refer to the same lines of code as before resequencing. If a line referred to in a statement does not exist, the line number is replaced with 32767.

If, because of the *initial-line* and *increment* chosen, the program requires lines larger than 32767, the resequencing process is halted and the program is unchanged.

Examples

RES resequences the lines of the program in memory to start with 100 and number by 10's.	`>RES`
RES 1000 resequences the lines of the program in memory to start with 1000 and number by 10's.	`>RES 1000`
RES 1000,15 resequences the lines of the program in memory to start with 1000 and number by 15's.	`>RES 1000,15`
RES ,15 resequences the lines of the program in memory to start with 100 and number by 15's.	`>RES ,15`

RESTORE

Format

RESTORE [*line-number*]

RESTORE #*file-number* [,REC *record-number*]

Description

The RESTORE statement can be used either with DATA statements or with files. When used with DATA statements, RESTORE sets the DATA statement which will be used by the next READ statement. If no *line-number* is given, the DATA statement with the lowest numbered line is used by the next READ statement. If *line-number* is given, then the DATA statement with that line number or (if it is not a DATA statement) the next DATA statement following it is used.

When used with files, the RESTORE statement sets the record that is used by the next PRINT, INPUT, or LINPUT statement referring to *file-number*. If no REC clause is given, the next record is the first record in the file, record number 0. If the REC clause is present, *record-number* specifies the next record to be used.

If there is pending output because of a previous PRINT, DISPLAY, PRINT...USING, or DISPLAY...USING, then that pending record is written on the file before the RESTORE statement is executed. Pending input data is removed by the RESTORE statement.

Examples

RESTORE sets the next DATA statement to be used to the first DATA statement in the program.

```
>100 RESTORE
```

RESTORE 130 sets the next DATA statement to be used to the DATA statement at line 130 or, if line 130 is not a DATA statement, to the next DATA statement after line 130.

```
>100 RESTORE 130
```

RESTORE #1 sets the next record to be used by the next PRINT, INPUT, or LINPUT statement using file #1 to be the first record in the file.

```
>100 RESTORE #1
```

RESTORE #4,REC H5 sets the next record to be used by the next PRINT, INPUT, or LINPUT statement using file #4 to be record H5.

```
>100 RESTORE #4,REC H5
```


RETURN (with GOSUB)

Format

RETURN

Description

See also RETURN (with ON ERROR).

RETURN used with GOSUB transfers control back to the statement after the GOSUB or ON...GOSUB which was most recently executed.

Program

The program on the right illustrates a use of RETURN as used with GOSUB. The program figures interest on an amount of money put in savings.

```
>100 CALL CLEAR
>110 INPUT "AMOUNT DEPOSITED:
 ":AMOUNT
>120 INPUT "ANNUAL INTEREST R
 ATE: ":RATE
>130 IF RATE<1 THEN RATE=RATE
 *100
>140 PRINT "NUMBER OF TIMES C
 OMPOUNDED"
>150 INPUT "ANNUALLY: ":COMP
>160 INPUT "STARTING YEAR: ":
 Y
>170 INPUT "NUMBER OF YEARS:
 ":N
>180 CALL CLEAR
>190 FOR A=Y TO Y+N
>200 GOSUB 240
>210 PRINT A,INT(AMOUNT*100+.
 5)/100
>220 NEXT A
>230 STOP
>240 FOR B=1 TO COMP
>250 AMOUNT=AMOUNT+AMOUNT*RAT
 E/(COMP*100)
>260 NEXT B
>270 RETURN
```

RETURN (with ON ERROR)

Format

RETURN [*line-number*]
RETURN [NEXT]

Description

See also RETURN (with GOSUB).

RETURN is used with ON ERROR. After an ON ERROR statement has been executed, an error causes transfer to the line specified in the ON ERROR statement. That line, or one after it, should be a RETURN statement. If RETURN is given without anything following it, control is returned to the statement on which the error occurred and the program executes it again. If RETURN is followed by *line-number*, control is transferred to the line specified and execution starts with that line. If RETURN is followed by NEXT, control is transferred to the statement following the one that caused the error.

Program

The program on the right illustrates the use of RETURN with ON ERROR. Line 120 causes an error to transfer control to line 170. Line 130 causes an error. Line 140, the next line after the one that causes the error, prints 140. Line 170 checks to see if the error has occurred four times and transfers control to 220 if it has. Line 180 increments the error counter by one. Line 190 prints 190. Line 200 resets the error handling to transfer to line 170. Line 210 returns to the line that caused the error and executes it again. Line 220, which is executed only after the error has occurred four times, prints 220 and returns to the line following the line that caused the error.

```
>100 CALL CLEAR
>110 A=1
>120 ON ERROR 170
>130 X=VAL("D")
>140 PRINT 140
>150 STOP
>160 REM ERROR HANDLING
>170 IF A>4 THEN 220
>180 A=A+1
>190 PRINT 190
>200 ON ERROR 170
>210 RETURN
>220 PRINT 220 :: RETURN NEXT
RUN
 190
 190
 190
 190
 220
 140
```

Also see the example of the ON ERROR statement.

RND

Format

RND

Description

The RND function returns the next pseudo-random number in the current sequence of pseudo-random numbers. The number returned is greater than or equal to zero and less than one. The sequence of random numbers returned is the same every time a program is run unless the RANDOMIZE statement appears in the program.

Examples

COLOR16 = INT(RND* 16) + 1 sets COLOR16 equal to some number from 1 through 16.

```
>100 COLOR16=INT(RND*16)+1
```

VALUE = INT(RND* 16) + 10 sets VALUE equal to some number from 10 through 25.

```
>100 VALUE=INT(RND*16)+10
```

LL(8) = INT(RND*(B − A + 1)) + A sets LL(8) equal to some number from A through B.

```
>100 LL(8)=INT(RND*(B-A+1))+A
```

RPT$

Format

RPT$(*string-expression*,*numeric-expression*)

Description

The RPT$ function returns a string equal to *numeric-expression* repetitions of *string-expression*. If RPT$ produces a string longer than 255 characters, the excess characters are discarded and a warning is given.

Examples

M$ = RPT$("ABCD",4) sets M$ equal to "ABCDABCDABCDABCD".	`>100 M$=RPT$("ABCD",4)`
CALL CHAR(96,RPT$("0000FFFF",8)) defines characters 96 through 99 with the string "0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF".	`>100 CALL CHAR(96,RPT$("0000FFFF",8))`
PRINT USING:RPT$("#",40):X$ prints the value of X$ using an image that consists of 40 number signs.	`>100 PRINT USING RPT$("#",40):X$`

RUN

Format

RUN ["*device.program-name*"]

RUN [*line-number*]

Description

The RUN command, which can also be used as a statement, starts program execution. The program to be run is first loaded into memory from *device.program-name* if that option is specified. The program is then checked for certain errors, such as FOR-NEXT loops that are missing the NEXT statement, and errors in syntax in statements. The values of all numeric variables are set to zero and the values of all string variables are set to null (a string of zero characters). The program is then executed.

Options

If *device.program-name* is specified, the program to be run is loaded from the specified device. The program and data currently in memory are lost.

If *line-number* is specified, the program in memory is run starting at *line-number*.

Examples

RUN causes the computer to begin execution of the program in memory.	`>RUN`
RUN 200 causes the computer to begin execution of the program in memory starting at line 200.	`>RUN 200` `>100 RUN 200`
RUN "DSK1.PRG3" causes the computer to load and begin execution of the program named PRG3 from the diskette in disk drive 1.	`>RUN "DSK1.PRG3"` `>320 RUN "DSK1.PRG3"`

Program

The program at the right illustrates the use of the RUN command used as a statement. It creates a "menu" and lets the person using the program chose what other program he wishes to run. The other programs should RUN this program rather than ending in the usual way, so that the menu is given again after they are finished.

```
>100 CALL CLEAR
>110 PRINT "1 PROGRAM 1."
>120 PRINT "2 PROGRAM 2."
>130 PRINT "3 PROGRAM 3."
>140 PRINT "4 END."
>150 PRINT
>160 INPUT "YOUR CHOICE: ":C
>170 IF C=1 THEN RUN "DSK1.PR
 G1"
>180 IF C=2 THEN RUN "DSK1.PR
 G2"
>190 IF C=3 THEN RUN "DSK1.PR
 G3"
>200 IF C=4 THEN STOP
>210 GOTO 100
```


SAVE

Format

SAVE *device.program-name* [,PROTECTED]
SAVE *device.program-name* [,MERGE]

Description

The SAVE command allows you to copy the program in memory to an external *device* under the name *program-name*. By using the OLD command, you can later recall the program into memory. The method of saving onto a cassette recorder is given in the *User's Reference Guide*. The method of saving onto a diskette is given in the *Disk Memory System* manual. SAVE clears breakpoints that have been put into a program.

Options

Only the PROTECTED option is available with cassette recorders.

By using the keyword PROTECTED, you may optionally specify that a program can only be run or brought into memory with OLD. The program cannot be listed, edited, or saved. This is not the same as using the protection available with the Disk Manager Module. NOTE: Be sure to keep an unprotected copy of any program because the protection feature is not reversable. If you also wish to protect the program from being copied, use the protect feature of the Disk Manager module.

You may optionally specify that the program is to be available for merging with another program by using the key word MERGE. Only programs saved with the key word MERGE may be merged with another program.

Examples

SAVE DSK1.PRG1 saves the program in memory on the diskette in disk drive 1 under the name PRG1.

```
>SAVE DSK1.PRG1
```

SAVE DSK1.PRG1,PROTECTED saves the program in memory on the diskette in disk drive 1 under the name PRG1. The program may be loaded into memory and run, but it may not be edited, listed, or resaved.

```
>SAVE DSK1.PRG1,PROTECTED
```

SAVE DSK1.PRG1,MERGE saves the program in memory on the diskette in disk drive 1 under the name PRG1. The program may later be merged with a program in memory by using the MERGE command.

```
>SAVE DSK1.PRG1,MERGE
```

SAY subprogram

Format

CALL SAY(*word-string* [,*direct-string*] [,...])

Description

The SAY subprogram causes the computer to speak *word-string* or the value specified by *direct-string* when the *Solid State Speech*™ Synthesizer (sold separately) is connected. For a complete description of SAY, see the manual that comes with the Speech Editor Command Module and Speech Synthesizer (both sold separately).

The value of *word-string* is any string value listed in *Appendix L*. If it is given as a literal value, it must be enclosed in quotation marks. The value of *direct-string* is a value returned by SPGET. The value of *direct-string* may be altered to add suffixes as described in *Appendix M*.

Word-strings and *direct-strings* must be alternated in the CALL SAY subprogram. If you wish to have two *direct-strings* or *word-strings* spoken consecutively, you may put in an extra comma to indicate the position of the item omitted.

Examples

CALL SAY("HELLO, HOW ARE YOU") causes the computer to say "Hello, how are you."

```
>100 CALL SAY("HELLO, HOW ARE
 YOU")
```

CALL SAY(,A$,,B$) causes the computer to say the the words indicated by A$ and B$, which must have been returned by SPGET.

```
CALL SAY(,A$,,B$)
```

Program

The program on the right illustrates using CALL SAY with a *word-string* and three *direct-strings*.

```
>100 CALL SPGET("HOW",X$)
>110 CALL SPGET("ARE",Y$)
>120 CALL SPGET("YOU",Z$)
>130 CALL SAY("HELLO",X$,,Y$,
 ,Z$)
```


SCREEN subprogram

Format

CALL SCREEN(*color-code*)

Description

The SCREEN subprogram changes the color of the screen to the color specified by *color-code*. All portions of the screen that do not have characters on them, or have characters or portions of characters that are color 1 (transparent), are shown as the color specified by *color-code*. The standard screen color for TI Extended BASIC is 8, cyan.

The color codes are:

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

Examples

CALL SCREEN(8) changes the screen to cyan, which is the standard screen color.

```
>100 CALL SCREEN(8)
```

CALL SCREEN(2) changes the screen to black.

```
>100 CALL SCREEN(2)
```

SEG$

Format

SEG$(*string-expression,position,length*)

Description

The SEG$ function returns a substring of a string. The string returned starts at *position* in *string-expression* and extends for *length* characters. If *position* is beyond the end of *string-expression*, the null string ("") is returned. If *length* extends beyond the end of *string-expression*, only the characters to the end are returned.

Examples

X$=SEG$("FIRSTNAME LASTNAME",1,9) sets X$ equal to "FIRSTNAME".

```
>100 X$=SEG$("FIRSTNAME LASTN
 AME",1,9)
```

Y$=SEG$("FIRSTNAME LASTNAME",11,8) sets Y$ equal to "LASTNAME".

```
>100 Y$=SEG$("FIRSTNAME LASTN
 AME",11,8)
```

Z$=SEG$("FIRSTNAME LASTNAME",10,1) sets Z$ equal to " ".

```
>100 Z$=SEG$("FIRSTNAME LASTN
 AME",10,1)
```

PRINT SEG$(A$,B,C) prints the substring of A$ starting at character B and extending for C characters.

```
>100 PRINT SEG$(A$,B,C)
```


SGN

Format

SGN(*numeric-expression*)

Description

The SGN function returns 1 if *numeric-expression* is positive, 0 if it is zero, and -1 if it is negative.

Examples

IF SGN(X2)=1 THEN 300 ELSE 400 transfers control to line 300 if X2 if positive and to line 400 if X2 is zero or negative.

```
>100 IF SGN(X2)=1 THEN 300 EL
 SE 400
```

ON SGN(X)+2 GOTO 200,300,400 transfers control to line 200 if X is negative, line 300 if X is zero, and line 400 if X is positive.

```
>100 ON SGN(X)+2 GOTO 200,300
 ,400
```

SIN

Format

SIN(*radian-expression*)

Description

The sine function gives the trigonometric sine of *radian-expression*. If the angle is in degrees, multiply the number of degrees by PI/180 to get the equivalent angle in radians.

Program

The program on the right gives the sine of several angles.

```
>100 A=.5235987755982
>110 B=30
>120 C=45*PI/180
>130 PRINT SIN(A);SIN(B)
>140 PRINT SIN(B*PI/180)
>150 PRINT SIN(C)
>RUN
 .5  -.9880316241
 .5
 .7071067812
```


SIZE

Format

SIZE

Description

The SIZE command displays the number of unused bytes of memory left in the computer. If the Memory Expansion peripheral is attached, the number of bytes available is given as the amount of stack free and the amount of program space free. A byte is the memory space required for such things as one character or digit, or one TI Extended BASIC keyword.

If the Memory Expansion is not attached, the space available is the amount of space left after the space taken up by the program, screen, character pattern definitions, sprite tables, color tables, string values, and the like is subtracted.

If the Memory Expansion is attached, the space available in the stack is the amount of space left after the space taken up by string values, information about variables, and the like is subtracted. Program space is the amount of space left after the space taken up by the program and the values of numeric variables is subtracted.

Examples

SIZE gives the available memory.

```
>SIZE
   13928 BYTES FREE
```

SIZE gives the available memory. If the Memory Expansion peripheral is attached, stack and program space are given.

```
>SIZE
   13928 BYTES OF STACK FREE
   24511 BYTES OF PROGRAM
         SPACE FREE
```

SOUND subprogram

Format

CALL SOUND(*duration,frequency1,volume1* [, ...*frequency4,volume4*])

Description

The SOUND subprogram tells the computer to produce tones or noise. The values given control three aspects of the sound: *Duration*; *frequency*; and *volume*.

Value	*Range*	*Description*
Duration	1 to 4250 −1 to −4250	The length of the sound in thousandths of a second.
Frequency	(Tone) 110 to 44733 (Noise) −1 to −8	What sound is played.
Volume	0 to 30	How loud the sound is.

Duration is from .001 to 4.250 seconds, although it may vary up to 1/60th of a second. The computer continues performing program statements while a sound is being played. When you call the SOUND subprogram, the computer waits until the previous sound has been completed before performing the new CALL SOUND. However, if a negative *duration* is specified, the previous sound is stopped and the new one is begun immediately.

Frequency specifies the frequency of the note to be played with a value from 110 to 44733. (NOTE: This range goes higher than the range of human hearing. People vary in their ability to hear high notes, but generally the highest is approximately a value of 10000.) The actual frequency produced by the computer may vary up to 10 percent. *Appendix D* lists the frequencies of some common notes.

A value of −1 to −8 specifies one of eight different types of noise.

Frequency	*Description*
−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 played simultaneously.

Volume specifies the loudness of the note or noise. Zero is loudest and 30 is softest.

Examples

CALL SOUND(1000,110,0) plays A below low C loudly for one second.

```
>100 CALL SOUND(1000,110,0)
```

CALL SOUND(500,110,0,131,0,196,3) plays A below low C and low C loudly, and G below C not as loudly, all for half a second.

```
>100 CALL SOUND(500,110,0,131
 ,0,196,3)
```

CALL SOUND(4250,−8,0) plays loud white noise for 4.250 seconds.

```
>100 CALL SOUND(4250,-8,0)
```

CALL SOUND(DUR,TONE,VOL) plays the tone indicated by TONE for a duration indicated by DUR, at a volume indicated by VOL.

```
>100 CALL SOUND(DUR,TONE,V
 OL)
```

Program

The program on the right plays the 13 notes of the first octave that is available on the computer.

```
>100 X=2^(1/12)
>110 FOR A=1 TO 13
>120 CALL SOUND(100,110*X^A,0
 )
>130 NEXT A
```

SPGET subprogram

Format

CALL SPGET(*word-string,return-string*)

Description

The SPGET subprogram returns in *return-string* the speech pattern that corresponds to *word-string*. For a complete description of SPGET, see the manual that comes with the Speech Editor Command Module and *Solid State Speech*™ Synthesizer (both sold separately).

The value of *word-string* is any string value listed in *Appendix L*. If it is given as a literal value, it must be enclosed in quotation marks. The value of *return-string* is used with SAY, and may be altered to add suffixes as described in *Appendix M*.

Program

The program on the right illustrates using CALL SPGET.

```
>100 CALL SPGET("HOW",X$)
>110 CALL SPGET("ARE",Y$)
>120 CALL SPGET("YOU",Z$)
>130 CALL SAY("HELLO",X$,,Y$,
 ,Z$)
```


SPRITE subprogram

Format

CALL SPRITE(*#sprite-number,character-value,sprite-color,dot-row*,dot-column, [*,row-velocity,column-velocity*] [,...])

Description

The SPRITE subprogram creates sprites. Sprites are graphics which have a color and a location anywhere on the screen. They can be set in motion in any direction at a variety of speeds, and continue their motion until it is changed by the program or the program stops. They move more smoothly than the usual character which jumps from one screen position to another.

Sprite-number is a numeric expression from 1 to 28. If the value is that of a sprite that is already defined, the old sprite is deleted and replaced by the new sprite. If the old sprite has a *row-* or *column-velocity*, and no new one is specified, the new sprite retains the old *velocities*.

Sprites pass over fixed characters on the screen. When two or more sprites are coincident, the sprite with the lowest sprite number covers the other sprites. While five or more sprites are on the same screen row, the one(s) with the highest sprite number(s) disappear.

Character-value may be any integer from 32 to 143. See the CHAR subprogram for information on defining characters. The *character-value* can be changed by the PATTERN subprogram. The sprite is defined as the character given and, in the case of double-sized sprites, the next three characters. See the MAGNIFY subprogram for more information.

Sprite-color may be any numeric expression from 1 to 16. It determines the foreground color of the sprite. The background color of a sprite is always 1, transparent. See the COLOR and SCREEN subprograms for more information.

Dot-row and *dot-column* are numbered consecutively starting with 1 in the upper left hand corner of the screen. *Dot-row* can be from 1 to 192 and *dot-column* can be from 1 to 256. (Actually *dot-row* can go up to 256, but the positions from 193 through 256 are off the bottom of the screen.) The position of the sprite is the upper left hand corner of the character(s) which define it.

Information about the position of a sprite can be found using the POSITION, COINC, and DISTANCE subprograms. The location of a sprite can be changed using the LOCATE subprogram. COLOR changes the color of a sprite. Sprites can be deleted with the DELSPRITE subprogram.

When a breakpoint occurs or the program stops, sprites cease to exist. They do not reappear with CONTINUE.

Options

Row-velocity and *column-velocity* may optionally be specified when the sprite is created. If both *row-* and *column-velocity* are zero, the sprite is stationary. A positive *row-velocity* moves the sprite down and a negative value moves it up. A positive *column-velocity* moves the sprite to the right and a negative value moves it to the left. If both *row-velocity* and *column-velocity* are non-zero, the sprite moves at an angle in a direction determined by the actual values.

Row- and *column-velocity* may be from – 128 to 127. A value close to zero is very slow. A value far from zero is very fast. When a sprite comes to the edge of the screen, it disappears and reappears in the corresponding position on the other side of the screen. The velocity of a sprite may be changed using the MOTION subprogram.

Programs

The following three programs show some possible uses of sprites. The third one uses all the subprograms that can relate to sprites except for COLOR and DISTANCE.

```
>100 CALL CLEAR
>110 CALL CHAR(96,"FFFFFFFFFF
 FFFFFF")
>120 CALL CHAR(98,"183C7EFFFF
 7E3C18")
>130 CALL CHAR(100,"F00FF00FF
 00FF00F")
```

Line 140 creates a dark blue sprite in the center of the screen and a dark red sprite in the upper left corner of the screen. Line 150 creates a white sprite near the upper right corner of the screen and starts it moving slowly at a 45 degree angle down and to the right. The sprite is an exclamation point.

```
>140 CALL SPRITE(#1,96,5,92,1
 24,#2,100,7,1,1)
>150 CALL SPRITE(#28,33,16,12
 ,48,1,1)
```

Line 160 creates a sprite at the upper left corner of the screen and starts it moving very fast at a 45 degree angle up and to the right.

```
>160 CALL SPRITE(#15,98,14,1,
 1,127,-128)
>170 GOTO 170
 (Press SHIFT C to stop the
 program.)
```


The program on the right makes a rather spectacular use of sprites. Line 110 defines character 96. Line 150 defines the sprites, 28 in all. The *sprite-number* is the current value of A. The *character-value* is 96. The *sprite-color* is INT(A/3) + 3. The starting *dot-row* and *dot-column* are 92 and 124, the center of the screen. The *row-* and *column-velocities* are chosen randomly using the value of A*INT(RND*4.5) – 2.25 + A/2*SGN(RND – .5). Line 170 causes the sequence to repeat.

```
>100 CALL CLEAR
>110 CALL CHAR(96,"0008081C7F
 1C0808")
>120 RANDOMIZE
>130 CALL SCREEN(2)
>140 FOR A=1 TO 28
>150 CALL SPRITE(#A,96,INT(A/
 3)+3,92,124,A*INT(RND*4.5)-2
 .25+A/2*SGN(RND-.5),A*INT(RN
 D*4.5)-2.25+A/2*SGN(RND-.5))
>160 NEXT A
>170 GOTO 140
 (Press SHIFT C to stop the
 program.)
```

The following program uses all the subprograms that can relate to sprites except for COLOR and DISTANCE. They are CHAR, COINC, DELSPRITE, LOCATE, MAGNIFY, MOTION, PATTERN, POSITION, and SPRITE.

The program creates two double sized magnified sprites in the shape of a person, walking along a floor. There is a barrier that one of them passes through and the other jumps through. The one that jumps through goes a little faster after each jump, so eventually it catches the other one. When it does, they each become double size *un*magnified sprites and continue walking. When they meet the second time, the one that has been going faster disappears and the other continues walking.

Lines 110, 120, 140, 150, 250, and 260 define the sprites.

```
>100 CALL CLEAR
>110 S1$="0103030103030303030
 303030303030380C0C080C0C0C0C
 0C0C0C0C0C0C0C0E0"
>120 S2$="0103030103070F1B1B0
 30303060C0C0E80C0C080C0E0F0D
 8CCC0C0C060303038"
```

Line 130 sets the meeting counter to zero.

```
>130 COUNT=0
>140 CALL CHAR(96,S1$)
>150 CALL CHAR(100,S2$)
>160 CALL SCREEN(14)
```

Lines 170 through 200 build the floor.

```
>170 CALL COLOR(14,13,13)
>180 FOR A=19 TO 24
>190 CALL HCHAR(A,1,136,32)
>200 NEXT A
```

Lines 210 through 240 build the barrier.

```
>210 CALL COLOR(13,15,15)
>220 CALL VCHAR(14,22,128,6)
>230 CALL VCHAR(14,23,128,6)
>240 CALL VCHAR(14,24,128,6)
>250 CALL SPRITE(#1,96,5,113
 ,129,#2,96,7,113,9)
>260 CALL MAGNIFY(4)
```

Line 270 sets the starting speed of the sprite that will speed up.

```
>270 XDIR=4
>280 PAT=2
```

Line 290 sets the sprites in motion.

```
>290 CALL MOTION(#1,0,XDIR,#2
 ,0,4)
```

Line 300 creates the illusion of walking.

```
>300 CALL PATTERN(#1,98+PAT,#
 2,98-PAT)
>310 PAT=-PAT
```

Line 320 checks to see if the sprites have met.

```
>320 CALL COINC(ALL,CC)
```

Line 330 transfers control if the sprites have met. Lines 340 and 350 check to see if the sprite has reached the barrier and transfer control if it has.

```
>330 IF CC<>0 THEN 370
>340 CALL POSITION(#1,YPOS1,X
 POS1)
>350 IF XPOS1>136 AND XPOS1<1
 92 THEN 470
```

Line 360 loops back to continue the walk. Lines 370 through 460 handle the sprites running into each other. Lines 380 and 390 stop them.

```
>360 GOTO 300
>370 REM COINCIDENCE
>380 CALL MOTION(#1,0,0,#2,0,
 0)
>390 CALL PATTERN(#1,96,#2,96
 )
```

Line 400 checks to see if it is the first meeting. Line 410 increments the meeting counter. Line 420 finds their position.

```
>400 IF COUNT>0 THEN 540
>410 COUNT=COUNT+1
>420 CALL POSITION(#1,YPOS1,X
 POS1,#2,YPOS2,XPOS2)
```

Line 430 makes them smaller. Line 440 puts them on the floor and moves the fast one slightly ahead.

```
>430 CALL MAGNIFY(3)
>440 CALL LOCATE(#1,YPOS1+16,
 XPOS1+8,#2,YPOS2+16,XPOS2)
```

Line 450 starts them moving again.

```
>450 CALL MOTION(#1,0,XDIR,#2
 ,0,4)
>460 GOTO 340
```


Lines 470 through 530 handle the fast sprite jumping through the barrier. Line 480 stops it. Line 490 finds where it is.

```
>470 REM #1 HIT WALL
>480 CALL MOTION(#1,0,0)
>490 CALL POSITION(#1,YPOS1,X
 POS1)
```

Line 500 puts it at the new location beyond the barrier. Lines 510 and 520 start it moving again, a little faster.

```
>500 CALL LOCATE(#1,YPOS1,193
 )
>510 XDIR=XDIR+1
>520 CALL MOTION(#1,0,XDIR)
>530 GOTO 300
```

Lines 540 through 640 handle the second meeting.

```
>540 REM SECOND COINCIDENCE
>550 FOR DELAY=1 TO 500 :: NE
 XT DELAY
```

Line 560 starts the slow sprite moving, while line 570 deletes the fast sprite. Lines 580 through 630 make the slow sprite walk 20 steps.

```
>560 CALL MOTION(#2,0,4)
>570 CALL DELSPRITE(#1)
>580 FOR STEP1=1 TO 20
>590 CALL PATTERN(#2,100)
>600 FOR DELAY=1 TO 20 :: NEX
 T DELAY
>610 CALL PATTERN(#2,96)
>620 FOR DELAY=1 TO 20 :: NEX
 T DELAY
>630 NEXT STEP1
>640 CALL CLEAR
```

SQR

Format

SQR(*numeric-expression*)

Description

The SQR function returns the positive square root of *numeric-expression*. SQR(X) is equivalent to X∧(1/2). *Numeric-expression* may not be a negative number.

Examples

PRINT SQR(4) prints 2.

```
>100 PRINT SQR(4)
```

X = SQR(2.57E5) sets X equal to the square root of 257,000 which is 506.9516742.

```
>100 X=SQR(2.57E5)
```

STOP

Format

STOP

Description

The STOP statement stops program execution. It can be used interchangeably with the END statement except that it may not be placed after subprograms.

Program

The program on the right illustrates the use of the STOP statement. The program adds the numbers from 1 to 100.

```
>100 CALL CLEAR
>110 TOT=0
>120 NUMB=1
>130 TOT=TOT+NUMB
>140 NUMB=NUMB+1
>150 IF NUMB>100 THEN PRINT T
 OT::STOP
>160 GOTO 130
```


STR$

Format

STR$(*numeric-expression*)

Description

The STR$ function returns a string equivalent to *numeric-expression*. This allows the functions, statements, and commands that act on strings to be used on the character representation of *numeric-expression*. The STR$ function is the inverse of the VAL function.

Examples

NUM$ = STR$(78.6) sets NUM$ equal to "78.6".

```
>100 NUM$=STR$(78.6)
```

LL$ = STR$(3E15) sets LL$ equal to "3.E15".

```
>100 LL$=STR$(3E15)
```

I$ = STR$(A*4) sets I$ equal to a string equal to what ever value is obtained when A is multiplied by 4. For instance, if A is equal to −8, I$ is set equal to "−32".

```
>100 I$=STR$(A*4)
```

SUB

Format

SUB *subprogram-name* [(*parameter-list*)]

Description

The SUB statement is the first statement in a subprogram. Subprograms are used when you wish to separate a group of statements from the main program. You may use subprograms to perform an operation several times in a program or in several different programs or to use variables that are specific to the subprogram. The SUB statement may not be in an IF-THEN-ELSE statement.

Subprograms are called with CALL *subprogram-name* [(*parameter-list*)]. Subprograms are ended with SUBEND, and left when either a SUBEND or a SUBEXIT statement is executed. Control is returned to the statement following the statement that called the subprogram. You must never transfer control out of a subprogram with any statement except SUBEND or SUBEXIT. This includes passing control with ON ERROR.

When a subprogram is in a program, it must follow the main program. The structure of a program must be as follows:

Start of Main Program . . . Subprogram Calls . . .	
End of Main Program	*The program will stop here without a STOP or END statement.*
Start of First Subprogram . . .	*Subprograms are optional.*
End of First Subprogram	*Nothing may appear between subprograms except remarks and the END statement.*
Start of Second Subprogram . . .	
End of Second Subprogram	*Only remarks and END may appear after the subprograms.*
End of Program	

Options

All variables used in a subprogram other than those in *parameter-list* are local to that subprogram, so you may use the same variable names that are used in the main program or in other subprograms, and alter their values, without having any effect on other variables. Likewise, the values of variables in the main program or other subprograms have no effect on the values of the variables in the subprogram. (However, DATA statements are available to subprograms.)

Communicating values to and from the main program is done with the optional *parameter-list*. The parameters need not have the same names as in the calling statement, but they must be of the same data type (numeric or string) and in the same order as the items in the CALL. If simple variables passed to subprograms have their values changed in the subprogram, the values of the variables in the main program are also changed. An array element such as A(1) in the parameter list of the calling statement is also changed in value in the main program when control is returned to the main program.

A value that is given in the calling statement as an expression is passed as a value only and changes in the value in the subprogram do not change values in the main program. Entire arrays are passed by reference, so changes in elements in the subprogram also change the values of the elements of the array in the main program. Arrays are indicated by following the parameter name with parentheses. If the array has more than one dimension, a comma must be placed inside the parentheses for each additional dimension.

If you wish, you may pass values only for simple variables by enclosing them in parentheses. Then the value can be used in the subprogram, but it is not changed in the return to the main program. For example, CALL SPRG1((A)) passes the value of A to a subprogram that starts SUB SPRG1(X), and allows that value to be used in X, but does not change the value of A in the main program if the subprogram changes the value of X.

If a subprogram is called more than once, any local variables used in the subprogram retain those values from one call to the next.

Examples

SUB MENU marks the beginning of a subprogram. No parameters are passed or returned.

```
>100 SUB MENU
```

SUB MENU(COUNT,CHOICE) marks the beginning of a subprogram. The variables COUNT and CHOICE may be used and/or have their values changed in the subprogram and returned to the variables in the same position in the calling statement.

```
>100 SUB MENU(COUNT,CHOICE)
```

SUB PAYCHECK(DATE,Q,SSN, PAYRATE,TABLE(,)) marks the beginning of a subprogram. The variables DATE, Q, SSN, PAYRATE, and the array TABLE with two dimensions may be used and/or have their values changed in the subprogram and returned to the variables in the same position in the calling statement.

```
>100 SUB PAYCHECK(DATE,Q,SSN,
 PAYRATE,TABLE(,))
```


Program

The program on the right illustrates the use of SUB. The subprogram MENU had been previously saved with the merge option. It prints a menu and requests a choice. The main program tells the subprogram how many choices there are and what the choices are. It then uses the choice made in the subprogram to determine what program to run.

```
>100 CALL MENU(5,R)
>110 ON R GOTO 120,130,140,15
 0,160
>120 RUN "DSK1.PAYABLES"
>130 RUN "DSK1.RECEIVE"
>140 RUN "DSK1.PAYROLL"
>150 RUN "DSK1.INVENTORY"
>160 RUN "DSK1.LEDGER"
>170 DATA ACCOUNTS PAYABLE,AC
 COUNTS RECEIVABLE,PAYROLL,IN
 VENTORY,GENERAL LEDGER
```

Beginning of subprogram MENU.

```
>10000 SUB MENU(COUNT,CHOICE)
>10010 CALL CLEAR
>10020 IF COUNT>22 THEN PRINT
  "TOO MANY ITEMS" :: CHOICE=
 0 :: SUBEXIT
>10030 RESTORE
```

Note that this R is not the same as the R used in lines 100 and 110 in the main program.

```
>10040 FOR R=1 TO COUNT
>10050 READ TEMP$
>10060 TEMP$=SEG$(TEMP$,1,25)
>10070 DISPLAY AT(R,1):R;TEMP
 $
>10080 NEXT R
>10090 DISPLAY AT(R+1,1):"YOU
 R CHOICE: 1"
>10100 ACCEPT AT(R+1,14)BEEP
 VALIDATE(DIGIT)SIZE(-2):CHOI
 CE
>10110 IF CHOICE<1 OR CHOICE>
 COUNT THEN 10100
>10120 SUBEND
```

SUBEND

Format

SUBEND

Description

The SUBEND statement marks the end of a subprogram. When SUBEND is executed, control is passed to the statement following the statement that called the subprogram. The SUBEND statement must always be the last statement in a subprogram. The SUBEND statement may not be in an IF-THEN-ELSE statement. The only statements that may immediately follow a SUBEND statement are REM, END, or the SUB statement for the next subprogram.

SUBEXIT

Format

SUBEXIT

Description

The SUBEXIT statement allows leaving a subprogram before the end of the subprogram (indicated with SUBEND). When it is executed, control is passed to the statement following the statement that called the subprogram. The SUBEXIT statement need not be present in a subprogram.

TAB

Format

TAB(*numeric-expression*)

Description

The TAB function specifies the starting position for the next *print-item* in a PRINT, PRINT...USING, DISPLAY, or DISPLAY...USING statement. If *numeric-expression* is greater than the length of a record for the device on which the printing is being done (for example: 28 for the screen, 32 for the thermal printer, the specified value for a file on a diskette or cassette), then it is repeatedly reduced by the record length until it is between 1 and the record length.

If the number of characters already printed on the current record is less than or equal to *numeric-expression*, the next print item is printed beginning on the position indicated by *numeric-expression*. If the number of characters already printed on the current record is greater than the position indicated by *numeric-expression*, the next *print-item* is printed on the next record beginning in the position indicated by *numeric-expression*.

The TAB function is treated as a *print-item*, so it must have a print separator (colon, semicolon, or comma) before and/or after it. The print separator before TAB is evaluated before the TAB function. Normally semicolons are used before and after TAB.

Examples

PRINT TAB(12);35 prints the number 35 at the twelfth position.	`>100 PRINT TAB(12);35`
PRINT 356;TAB(18);"NAME" prints 356 at the beginning of the line and NAME at the eighteenth position of the line.	`>100 PRINT 356;TAB(18);"NAME"`
PRINT "ABCDEFGHIJKLM";TAB(5); "NOP" prints ABCDEFGHIJKLM at the beginning of the line and NOP at the fifth position of the next line.	`>100 PRINT "ABCDEFGHIJKLM";TA B(5);"NOP"`
DISPLAY AT(12,1):"NAME";TAB (15);"ADDRESS" displays NAME at the beginning of the twelfth line on the screen and ADDRESS at the fifteenth position on the twelfth line of the screen.	`>100 DISPLAY AT(12,1):"NAME"; TAB(15);"ADDRESS"`

TAN

Format

TAN(*radian-expression*)

Description

The tangent function gives the trigonometric tangent of *radian-expression*. If the angle is in degrees, multiply the number of degrees by PI/180 to get the equivalent angle in radians.

Program

The program on the right gives the tangent of several angles.

```
>100 A=.7853981633973
>110 B=26.565051177
>120 C=45*PI/180
>130 PRINT TAN(A);TAN(B)
>140 PRINT TAN(B*PI/180)
>150 PRINT TAN(C)
>RUN
 1.   7.17470553
 .5
 1
```

TRACE

Format

TRACE

Description

The TRACE command causes each line number to be displayed on the screen before the statements on that line are executed. This enables you to follow the course of a program as a debugging aid. The TRACE command may be used as a statement. The effect of the TRACE command is canceled when the NEW command or UNTRACE command or statement is performed.

Example

TRACE causes the computer to display a trace of the lines of a program on the screen.

```
>TRACE
>100 TRACE
```


UNBREAK

Format

UNBREAK [*line-list*]

Description

The UNBREAK command removes all breakpoints. It can optionally be set for only those in *line-list*. UNBREAK can be used as a statement.

Examples

UNBREAK removes all breakpoints.

```
>UNBREAK
>420 UNBREAK
```

UNBREAK 100,130 removes the breakpoints from lines 100 and 130.

```
>UNBREAK 100,130
>320 UNBREAK 100,130
```

UNTRACE

Format

UNTRACE

Description

The UNTRACE command removes the effect of the TRACE command. UNTRACE can be used as a statement.

Example

UNTRACE removes the effect of TRACE.

```
>UNTRACE
>420 UNTRACE
```

VAL

Format

VAL(*string-expression*)

Description

The VAL function returns the number equivalent to *string-expression*. This allows the functions, statements, and commands that act on numbers to be used on *string-expression*. The VAL function is the inverse of the STR$ function.

Examples

NUM = VAL("78.6") sets NUM equal to 78.6. >100 NUM=VAL("78.6")

LL = VAL("3E15") sets LL equal to 3.E15. >100 LL=VAL("3E15")

VCHAR

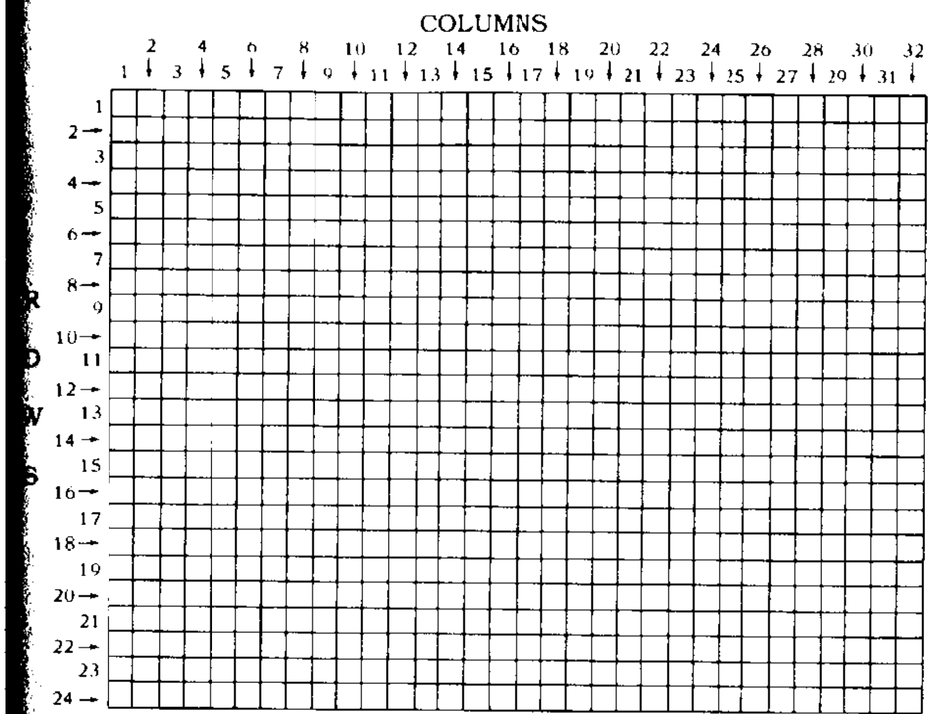
Format

CALL VCHAR(*row*,*column*,*character-code* [,*repetition*])

Description

The VCHAR subprogram places a character anywhere on the display screen and optionally repeats it vertically. The character with the ASCII value of *character-code* is placed in the position described by *row* and *column* and is repeated vertically *repetition* times.

A value of 1 for *row* indicates the top of the screen. A value of 24 is the bottom of the screen. A value of 1 for *column* indicates the left side of the screen. A value of 32 is the right side of the screen. The screen can be thought of as a grid as shown below.



Examples

CALL VCHAR(12,16,33) places character 33 (an exclamation point) in row 12, column 16. >100 CALL VCHAR(12,16,33)

CALL VCHAR(1,1,ASC("!"),768) places an exclamation point in row 1, column 1, and repeats it 768 times, which fills the screen. >100 CALL VCHAR(1,1,ASC("!"),768)

CALL VCHAR(R,C,K,T) places the character with an ASCII code of K in row R, column C and repeats it T times. >100 CALL VCHAR(R,C,K,T)

VERSION subprogram

Format

CALL VERSION(*numeric-variable*)

Description

The VERSION subprogram returns a value indicating the version of BASIC that is being used. TI Extended BASIC returns a value of 100.

Example

CALL VERSION(V) sets V equal to 100.

```
>100 CALL VERSION(V)
```


Appendices

The following appendices give useful information concerning TI Extended BASIC.

List of Illustrative Programs

ELEMENT ILLUSTRATED	LINES	DESCRIPTION	PAGE
	44	Codebreaker Game	27
ACCEPT	16	Entry of 20 names	48
CALL	8	CLEAR and user written subroutine	55
CHAR	12	1. Moving figure	58
	7	2. Resetting characters	58
CHR$	4	List of ASCII codes	60
CLEAR	3	(Simple example)	61
	3	(Simple example)	61
COINC	10	(Simple example)	65
COS	6	(Simple example)	69
DATA	14	(Simple example)	71
DELETE	2	(Simple example)	74
DISPLAY	18	Draw on screen	78
ERR	5	(Simple example)	84
FOR-TO-STEP	11	Design	87
GOSUB	24	Probability	90
GOTO	8	Add 1 through 100	91
IF-THEN-ELSE	17	Sequence numbers	96
IMAGE	12	(Simple example)	99
	2	(Simple example)	100
INPUT	17	Writes letter	103
INPUT (with files)	12	(Simple example)	106
JOYST	5	Moves sprite	108
KEY	12	Moves sprite	109
LINPUT	6	(Simple example)	113
LOCATE	6	(Simple example)	116
LOG	8	Log to any base	117

ELEMENT ILLUSTRATED	LINES	DESCRIPTION	PAGE
MAGNIFY	17	(Simple example)	120
MERGE	13	Moves sprite	122
MOTION	8	Moves sprite	125
NEXT	6	(Simple example)	127
NUMBER	4	(Simple example)	128
ON BREAK	11	(Simple example)	130
ON ERROR	15	(Simple example)	132
ON...GOSUB	20	Choose with a menu	134
ON...GOTO	19	Choose with a menu	136
ON WARNING	8	(Simple example)	137
PATTERN	18	Rolling wheel	142
POS	8	Breakup sentence	145
PRINT	7	(Simple example)	149
RANDOMIZE	5	(Simple example)	151
REC	12	(Simple example)	153
RETURN (with GOSUB)	18	Figure interest	157
RETURN (with ON ERROR)	13	Handle error	158
RUN	12	Choose with a menu	162
SAY	4	(Simple example)	164
SIN	6	(Simple example)	168
SOUND	4	Play first 13 notes	171
SPGET	4	(Simple example)	172
SPRITE	8	(Simple example)	174
	8	Creation of stars	175
	55	Walking sprites	175
STOP	7	Add 1 through 100	178
SUB	21	Choose with a menu	183
TAN	6	(Simple example)	186

Commands, Statements, and Functions

The following is a list of all TI Extended BASIC commands, statements, and functions. Commands are listed first; if a command can also be used as a statement, the letter "S" is listed to the right of the command. Commands that can be abbreviated have the acceptable abbreviations underlined. Next is a list of all TI Extended BASIC statements; those that can also be used as commands have a "C" after them. Finally, there is a list of all TI Extended BASIC functions.

TI Extended BASIC Commands

BREAK S	MERGE	SAVE
BYE	NUMBER	SIZE
CONTINUE	OLD	TRACE S
DELETE S	RESEQUENCE	UNBREAK S
LIST	RUN S	UNTRACE S

TI Extended BASIC Statements

ACCEPT C	CALL HCHAR C	OPTION BASE
CALL	IF THEN ELSE	CALL PATTERN C
CALL CHAR C	IMAGE	CALL PEEK C
CALL CHARPAT C	CALL INIT C	CALL POSITION C
CALL CHARSET C	INPUT	PRINT C
CALL CLEAR C	INPUT REC	PRINT USING C
CLOSE C	CALL JOYST C	RANDOMIZE C
CALL COINC C	CALL KEY C	READ C
CALL COLOR C	[LET] C	REM C
DATA	CALL LINK C	RESTORE C
DEF	LINPUT	RETURN
CALL DELSPRITE C	CALL LOAD C	CALL SAY C
DIM C	CALL LOCATE C	CALL SCREEN C
DISPLAY C	CALL MAGNIFY C	CALL SOUND C
DISPLAY USING C	CALL MOTION C	CALL SPGET C
CALL DISTANCE C	NEXT C	CALL SPRITE C
END	ON BREAK	STOP C
CALL ERR C	ON ERROR	SUB
FOR C	ON GOSUB	SUBEND
CALL GCHAR C	ON GOTO	SUBEXIT
GOSUB	ON WARNING	CALL VCHAR C
GOTO	OPEN C	CALL VERSION C

TI Extended BASIC Functions

ABS	LEN	SEG$
ASC	LOG	SGN
ATN	MAX	SIN
CHR$	MIN	SQR
COS	PI	STR$
EOF	POS	TAB
EXP	REC	TAN
INT	RND	VAL
	RPT$	

ASCII Codes

APPENDIX **C**

The following predefined characters may be printed or displayed on the screen.

ASCII CODE	*CHARACTER*	*ASCII CODE*	*CHARACTER*
30	(cursor)	63	? (question mark)
31	(edge character)	64	@ (at sign)
32	(space)	65	A
33	! (exclamation point)	66	B
34	" (quote)	67	C
35	# (number or pound sign)	68	D
36	$ (dollar)	69	E
37	% (percent)	70	F
38	& (ampersand)	71	G
39	' (apostrophe)	72	H
40	((open parenthesis)	73	I
41	) (close parenthesis)	74	J
42	* (asterisk)	75	K
43	+ (plus)	76	L
44	, (comma)	77	M
45	- (minus)	78	N
46	. (period)	79	O
47	/ (slash)	80	P
48	0	81	Q
49	1	82	R
50	2	83	S
51	3	84	T
52	4	85	U
53	5	86	V
54	6	87	W
55	7	88	X
56	8	89	Y
57	9	90	Z
58	: (colon)	91	[(open bracket)
59	; (semicolon)	92	\ (reverse slash)
60	< (less than)	93	] (close bracket)
61	= (equals)	94	^ (exponentiation)
62	> (greater than)	95	_ (underline)

The following key presses may also be detected by CALL KEY.

1	**SHIFT A** (AID)	3	**SHIFT F** (DEL)
4	**SHIFT G** (INS)	6	**SHIFT R** (REDO)
7	**SHIFT T** (ERASE)	8	**SHIFT S** (LEFT ARROW)
9	**SHIFT D** (RIGHT ARROW)	10	**SHIFT X** (DOWN ARROW)
11	**SHIFT E** (UP ARROW)	12	**SHIFT V** (CMD)
13	**ENTER**	14	**SHIFT W** (BEGIN)
15	**SHIFT Z** (BACK)		

Musical Tone Frequencies

APPENDIX **D**

The following table gives the 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 that the computer can produce, it can be helpful for programming music.

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♯, A♭	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

APPENDIX E

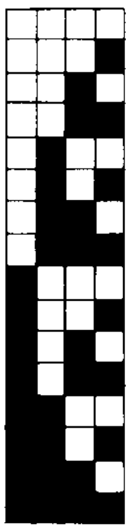
Character Sets

SET	ASCII CODES	SET	ASCII CODES
0	30-31		
1	32-39	8	88-95
2	40-47	9	96-103
3	48-55	10	104-111
4	56-63	11	112-119
5	64-71	12	120-127
6	72-79	13	128-135
7	80-87	14	136-143

APPENDIX F

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 G

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

Color Combinations

APPENDIX H

The following color combinations produce the sharpest, clearest character resolution.

BEST

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

SECOND BEST

2, 5	Black on Dark Blue	2, 11	Black on Dark Yellow
2, 4	Black on Light Green	2, 10	Black on Light Red
2, 12	Black on Light Yellow	13, 10	Dark Green on Light Red
13, 16	Dark Green on White	7, 16	Dark Red on White
6, 15	Light Blue on Gray	6, 4	Light Blue on Light Green
6, 16	Light Blue on White	4, 16	Light Green on White

THIRD BEST

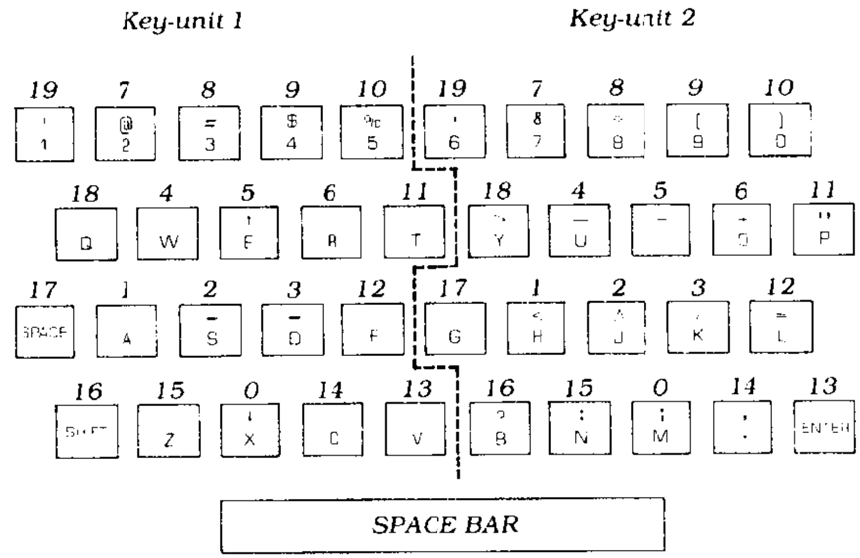
2, 16	Black on White	5, 12	Dark Blue on Light Yellow
7, 9	Dark Red on Medium Red	4, 12	Light Green on Light Yellow
14, 15	Magenta on Gray	14, 16	Magenta on White
3, 11	Medium Green on Dark Yellow	3, 15	Medium Green on Gray
9, 15	Medium Red on Gray	9, 10	Medium Red on Light Red
9, 12	Medium Red on Light Yellow	9, 16	Medium Red on White
16, 7	White on Dark Red		

FOURTH BEST

8, 2	Cyan on Black	8, 16	Cyan on White
7, 2	Dark Red on Black	7, 4	Dark Red on Light Green
15, 16	Gray on White	5, 2	Light Blue on Black
4, 2	Light Green on Black	10, 2	Light Red on Black
10, 16	Light Red on White	14, 12	Magenta on Light Yellow
9, 4	Medium Red on Light Green	16, 6	White on Light Blue

Split Console Keyboard

APPENDIX I



Character Codes for Split Keyboard

APPENDIX J

CODE	KEYS*	CODE	KEYS*
0	X, M	10	5, 0
1	A, H	11	T, P
2	S, J	12	F, L
3	D, K	13	V, ENT
4	W, U	14	C, .
5	E, I	15	Z, N
6	R, O	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.

Mathematical Functions

APPENDIX K

The following mathematical functions may be defined with DEF as shown.

Function	*TI Extended BASIC statement*
Secant	DEF SEC(X)=1/COS(X)
Cosecant	DEF CSC(X)=1/SIN(X)
Cotangent	DEF COT(X)=1/TAN(X)
Inverse Sine	DEF ARCSIN(X)=ATN(X/SQR(1-X*X))
Inverse Cosine	DEF ARCCOS(X)=-ATN(X/SQR(1-X*X))+PI/2
Inverse Secant	DEF ARCSEC(X)=ATN(SQR(X*X-1))+(SGN(X)-1)*PI/2
Inverse Cosecant	DEF ARCCSC(X)=ATN(1/SQR(X*X-1))+(SGN(X)-1)*PI/2
Inverse Cotangent	DEF ARCCOT(X)=PI/2-ATN(X) *or* =PI/2+ATN(-X)
Hyberbolic Sine	DEF SINH(X)=(EXP(X)-EXP(-X))/2
Hyberbolic Cosine	DEF COSH(X)=(EXP(X)+EXP(-X))/2
Hyperbolic Tangent	DEF TANH(X)=-2*EXP(-X)/(EXP(X)+EXP(-X))+1
Hyperbolic Secant	DEF SECH=2/(EXP(X)+EXP(-X))
Hyperbolic Cosecant	DEF CSCH=2/(EXP(X)-EXP(-X))
Hyperbolic Cotangent	DEF CCTH(X)=2*EXP(-X)/(EXP(X)-EXP(-X))+1
Inverse Hyperbolic Sine	DEF ARCSINH(X)=LOG(X+SQR(X*X+1))
Inverse Hyperbolic Cosine	DEF ARCCOSH(X)=LOG(X+SQR(X*X-1))
Inverse Hyperbolic Tangent	DEF ARCTANH(X)=LOG((1+X)/(1-X))/2
Inverse Hyperbolic Secant	DEF ARCSECH(X)=LOG((1+SQR(1-X*X))/X)
Inverse Hyperbolic Cosecant	DEF ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
Inverse Hyperbolic Cotangent	DEF ARCCOTH(X)=LOG((X+1)/(X-1))/2

List of Speech Words

APPENDIX L

The following is a list of all the letters, numbers, words, and phrases that can be accessed with CALL SAY and CALL SPGET. See *Appendix M* for instructions on adding suffixes to anything in this list.

- (NEGATIVE)
+ (POSITIVE)
. (POINT)
0
1
2
3
4
5
6
7
8
9

A (a)
A1 (ə)
ABOUT
AFTER
AGAIN
ALL
AM
AN
AND
ANSWER
ANY
ARE
AS
ASSUME
AT

B
BACK
BASE
BE
BETWEEN
BLACK
BLUE
BOTH
BOTTOM
BUT
BUY
BY
BYE

C
CAN
CASSETTE
CENTER
CHECK
CHOICE
CLEAR
COLOR
COME
COMES
COMMA
COMMAND
COMPLETE
COMPLETED
COMPUTER
CONNECTED
CONSOLE
CORRECT
COURSE
CYAN

D
DATA
DECIDE
DEVICE
DID
DIFFERENT
DISKETTE
DO
DOES
DOING
DONE
DOUBLE
DOWN
DRAW
DRAWING

E
EACH
EIGHT
EIGHTY
ELEVEN
ELSE
END
ENDS
ENTER
ERROR
EXACTLY
EYE

F
FIFTEEN
FIFTY
FIGURE
FIND
FINE
FINISH
FINISHED
FIRST
FIT
FIVE
FOR
FORTY
FOUR
FOURTEEN
FOURTH
FROM
FRONT

G
GAMES
GET
GETTING
GIVE
GIVES
GO
GOES
GOING
GOOD
GOOD WORK
GOODBYE
GOT
GRAY
GREEN
GUESS

H
HAD
HAND
HANDHELD UNIT
HAS
HAVE
HEAD
HEAR
HELLO
HELP

HERE
HIGHER
HIT
HOME
HOW
HUNDRED
HURRY
I
I WIN
IF
IN
INCH
INCHES
INSTRUCTION
INSTRUCTIONS
IS
IT
J
JOYSTICK
JUST
K
KEY
KEYBOARD
KNOW
L
LARGE
LARGER
LARGEST
LAST
LEARN
LEFT
LESS
LET
LIKE
LIKES
LINE
LOAD
LONG
LOOK
LOOKS
LOWER
M
MADE
MAGENTA
MAKE
ME
MEAN
MEMORY
MESSAGE
MESSAGES
MIDDLE
MIGHT
MODULE
MORE
MOST
MOVE
MUST
N
NAME
NEAR
NEED
NEGATIVE
NEXT
NICE TRY
NINE
NINETY
NO
NOT
NOW
NUMBER
O
OF
OFF
OH
ON
ONE
ONLY
OR
ORDER
OTHER
OUT
OVER
P
PART
PARTNER
PARTS
PERIOD
PLAY
PLAYS
PLEASE
POINT
POSITION
POSITIVE
PRESS
PRINT
PRINTER
PROBLEM
PROBLEMS
PROGRAM
PUT
PUTTING
Q
R
RANDOMLY
READ (read)
READ1 (red)
READY TO START
RECORDER
RED
REFER
REMEMBER
RETURN
REWIND
RIGHT
ROUND
S
SAID
SAVE
SAY
SAYS
SCREEN
SECOND
SEE
SEES
SET
SEVEN
SEVENTY
SHAPE
SHAPES
SHIFT
SHORT
SHORTER
SHOULD
SIDE
SIDES
SIX
SIXTY
SMALL
SMALLER
SMALLEST
SO
SOME
SORRY

SPACE
SPACES
SPELL
SQUARE
START
STEP
STOP
SUM
SUPPOSED
SUPPOSED TO
SURE
T
TAKE
TEEN
TELL
TEN
TEXAS INSTRUMENTS
THAN
THAT
THAT IS INCORRECT
THAT IS RIGHT
THE (the)
THE1 (thə)
THEIR
THEN
THERE
THESE
THEY
THING
THINGS
THINK
THIRD
THIRTEEN
THIRTY
THIS
THREE
THREW
THROUGH
TIME
TO
TOGETHER
TONE
TOO
TOP
TRY
TRY AGAIN
TURN
TWELVE
TWENTY
TWO
TYPE
U
UHOH
UNDER
UNDERSTAND
UNTIL
UP
UPPER
USE
V
VARY
VERY
W
WAIT
WANT
WANTS
WAY
WE
WEIGH
WEIGHT
WELL
WERE
WHAT
WHAT WAS THAT
WHEN
WHERE
WHICH
WHITE
WHO
WHY
WILL
WITH
WON
WORD
WORDS
WORK
WORKING
WRITE
X
Y
YELLOW
YES
YET
YOU
YOU WIN
YOUR
Z
ZERO

Adding Suffixes to Speech Words

This appendix describes how to add ING, S, and ED to any word available in the *Solid•State Speech*™ Synthesizer resident vocabulary.

The code for a word is first read using SPGET. The code consists of a number of characters, one of which tells the speech unit the length of the word. Then, by means of the subprograms listed here, additional codes can be added to give the sound of a suffix.

Words often have trailing-off data that make the word sound more natural but prevent the easy addition of suffixes. In order to add suffixes this trailing-off data must be removed.

The following program allows you to input a word and, by trying different truncation values, make the suffix sound like a natural part of the word. The subprograms DEFING (lines 1000 through 1130), DEFS1 (lines 2000 through 2100), DEFS2 (lines 3000 through 3090), DEFS3 (lines 4000 through 4120), DEFED1 (lines 5000 through 5070), DEFED2 (lines 6000 through 6110), DEFED3 (lines 7000 through 7130), and MENU (lines 10000 through 10120) should be input separately and saved with the MERGE option. (The subprogram MENU is the same one used in the illustrative program with SUB.) You may wish to use different line numbers. Each of these subprograms (except MENU) defines a suffix.

DEFING defines the ING sound. DEFS1 defines the S sound as it occurs at the end of "cats." DEFS2 defines the S sound as it occurs at the end of "cads." DEFS3 defines the S sound as it occurs at the end of "wishes." DEFED1 defines the ED sound as it occurs at the end of "passed." DEFED2 defines the ED sound as it occurs at the end of "caused." DEFED3 defines the ED sound as it occurs at the end of "heated."

In running the program, enter a 0 for the truncation value in order to leave the truncation sequence.

```
100 REM *******************
110 REM REQUIRES MERGE OF:
120 REM MENU (LINES 10000 THROUGH 10120)
130 REM DEFING (LINES 1000 THROUGH 1130)
140 REM DEFS1 (LINES 2000 THROUGH 2100)
150 REM DEFS2 (LINES 3000 THROUGH 3090)
160 REM DEFS3 (LINES 4000 THROUGH 4120)
170 REM DEFED1 (LINES 5000 THROUGH 5070)
180 REM DEFED2 (LINES 6000 THROUGH 6110)
190 REM DEFED3 (LINES 7000 THROUGH 7130)
200 REM *******************
210 CALL CLEAR
220 PRINT "THIS PROGRAM IS USED TO"
```



```
230 PRINT "FIND THE PROPER TRUNCATION"
240 PRINT "VALUE FOR ADDING SUFFIXES"
250 PRINT "TO SPEECH WORDS.": :
260 FOR DELAY=1 TO 300::NEXT DELAY
270 PRINT "CHOOSE WHICH SUFFIX YOU"
280 PRINT "WISH TO ADD.": :
290 FOR DELAY=1 TO 200::NEXT DELAY
300 CALL MENU(8,CHOICE)
310 DATA 'ING','S' AS IN CATS,'S' AS IN CADS,'S' AS IN WISHES,
'ED' AS IN PASSED,'ED' AS IN CAUSED,'ED' AS IN HEATED,END
320 IF CHOICE=0 OR CHOICE=8 THEN STOP
330 INPUT "WHAT IS THE WORD? ":WORD$
340 ON CHOICE GOTO 350,370,390,410,430,450,470
350 CALL DEFING(D$)
360 GOTO 480
370 CALL DEFS1(D$)!CATS
380 GOTO 480
390 CALL DEFS2(D$)!CADS
400 GOTO 480
410 CALL DEFS3(D$)!WISHES
420 GOTO 480
430 CALL DEFED1(D$)!PASSED
440 GOTO 480
450 CALL DEFED2(D$)!CAUSED
460 GOTO 480
470 CALL DEFED3(D$)!HEATED
480 REM TRY VALUES
490 CALL CLEAR
500 INPUT "TRUNCATE HOW MANY BYTES? ":L
510 IF L=0 THEN 300
520 CALL SPGET(WORD$,B$)
530 L=LEN(B$)-L-3
540 C$=SEG$(B$,1,2)&CHR$(L)&SEG$(B$,4,L)
550 CALL SAY(,C$&D$)
560 GOTO 500
```

The data has been given in short DATA statements to make it as easy as possible to input. It may be consolidated to make the program shorter.

```
1000 SUB DEFING(A$)
1010 DATA 96,0,52,174,30,65
1020 DATA 21,186,90,247,122,214
1030 DATA 179,95,77,13,202,50
1040 DATA 153,120,117,57,40,248
1050 DATA 133,173,209,25,39,85
1060 DATA 225,54,75,167,29,77
1070 DATA 105,91,44,157,118,180
1080 DATA 169,97,161,117,218,25
1090 DATA 119,184,227,222,249,238,1
1100 RESTORE 1010
1110 A$=""
1120 FOR I=1 TO 55::READ A::A$=A$&CHR$(A)::NEXT I
1130 SUBEND

2000 SUB DEFS1(A$)!CATS
2010 DATA 96,0,26
2020 DATA 14,56,130,204,0
2030 DATA 223,177,26,224,103
2040 DATA 85,3,252,106,106
2050 DATA 128,95,44,4,240
2060 DATA 35,11,2,126,16,121
2070 RESTORE 2010
2080 A$=""
2090 FOR I=1 TO 29::READ A::A$=A$&CHR$(A)::NEXT I
2100 SUBEND

3000 SUB DEFS2(A$)!CADS
3010 DATA 96,0,17
3020 DATA 161,253,158,217
3030 DATA 168,213,198,86,0
3040 DATA 223,153,75,128,0
3050 DATA 95,139,62
3060 RESTORE 3010
3070 A$=""
3080 FOR I=1 TO 20::READ A::A$=A$&CHR$(A)::NEXT I
3090 SUBEND
```



```
4000 SUB DEFS3(A$)!WISHES
4010 DATA 96,0,34
4020 DATA 173,233,33,84,12
4030 DATA 242,205,166,55,173
4040 DATA 93,222,68,197,188
4050 DATA 134,238,123,102
4060 DATA 163,86,27,59,1,124
4070 DATA 103,46,1,2,124,45
4080 DATA 138,129,7
4090 RESTORE 4010
4100 A$=""
4110 FOR I=1 TO 37::READ A::A$=A$&CHR$(A)::NEXT I
4120 SUBEND

5000 SUB DEFED1(A$)!PASSED
5010 DATA 96,0,10
5020 DATA 0,224,128,37
5030 DATA 204,37,240,0,0,0
5040 RESTORE 5010
5050 A$=""
5060 FOR I=1 TO 13::READ A::A$=A$&CHR$(A)::NEXT I
5070 SUBEND

6000 SUB DEFED2(A$)!CAUSED
6010 DATA 96,0,26
6020 DATA 172,163,214,59,35
6030 DATA 109,170,174,68,21
6040 DATA 22,201,220,250,24
6050 DATA 69,148,162,166,234
6060 DATA 75,84,97,145,204
6070 DATA 15
6080 RESTORE 6010
6090 A$=""
6100 FOR I=1 TO 29::READ A::A$=A$&CHR$(A)::NEXT I
6110 SUBEND
```

```
7000 SUB DEFED3(A$)!HEATED
7010 DATA 96,0,36
7020 DATA 173,233,33,84,12
7030 DATA 242,205,166,183
7040 DATA 172,163,214,59,35
7050 DATA 109,170,174,68,21
7060 DATA 22,201,92,250,24
7070 DATA 69,148,162,38,235
7080 DATA 75,84,97,145,204
7090 DATA 178,127
7100 RESTORE 7010
7110 A$=""
7120 FOR I=1 TO 39::READ A::A$=A$&CHR$(A)::NEXT I
7130 SUBEND

10000 SUB MENU(COUNT,CHOICE)
10010 CALL CLEAR
10020 IF COUNT>22 THEN PRINT "TOO MANY ITEMS" :: CHOICE=0 :: SUBEXIT
10030 RESTORE
10040 FOR I=1 TO COUNT
10050 READ TEMP$
10060 TEMP$=SEG$(TEMP$,1,25)
10070 DISPLAY AT(I,1):I;TEMP$
10080 NEXT I
10090 DISPLAY AT(I+1,1):"YOUR CHOICE: 1"
10100 ACCEPT AT(I+1,14)BEEP VALIDATE(DIGIT)SIZE(-2):CHOICE
10110 IF CHOICE<1 OR CHOICE>COUNT THEN 10100
10120 SUBEND
```


You can use the subprograms in any program once you have determined the number of bytes to truncate. The following program uses the subprogram DEFING in lines 1000 through 1130 to have the computer say the word DRAWING using DRAW plus the suffix ING. Note that it was found that DRAW should be truncated by 41 characters to produce the most natural sounding DRAWING. The subprogram DEFING in lines 1000 through 1130 is the program you saved with the merge option.

```
100 CALL DEFING(ING$)
110 CALL SPGET("DRAW",DRAW$)
120 L=LEN(DRAW$)-3-41! 3 BYTES OF SPEECH OVERHEAD, 41 BYTES TRUNCATED
130 DRAW$=SEG$(DRAW$,1,2)&CHR$(L)&SEG$(DRAW$,4,L)
140 CALL SAY("WE ARE",DRAW$&ING$,"A1 SCREEN")
150 GOTO 140
1000 SUB DEFING(A$)
1010 DATA 96,0,52,174,30,65
1020 DATA 21,186,90,247,122,214
1030 DATA 179,95,77,13,202,50
1040 DATA 153,120,117,57,40,248
1050 DATA 133,173,209,25,39,85
1060 DATA 225,54,75,167,29,77
1070 DATA 105,91,44,157,118,180
1080 DATA 169,97,161,117,218,25
1090 DATA 119,184,227,222,249,238,1
1100 RESTORE 1010
1110 A$=""
1120 FOR I=1 TO 55::READ A::A$=A$&CHR$(A)::NEXT I
1130 SUBEND
```

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

Errors

The following lists all the error messages that TI Extended BASIC gives. The first list is alphabetical by the message that is given, and the second list is numeric by the number of the error that is returned by CALL ERR. If the error occurs in the execution of a program, the error message is often followed by IN *line-number*.

Sorted by Message

#	Message	Descriptions of Possible Errors

74 **BAD ARGUMENT**
- Bad value given in ASC, ATN, COS, EXP, INT, LOG, SIN, SOUND, SQR, TAN, or VAL.
- An array element specified in a SUB statement.
- Bad first parameter or too many parameters in LINK.

61 **BAD LINE NUMBER**
- Line number less than 1 or greater than 32767.
- Omitted line number.
- Line number outside the range 1 through 32767 produced by RES.

57 **BAD SUBSCRIPT**
- Use of too large or small subscript in an array.
- Incorrect subscript in DIM.

79 **BAD VALUE**
- Incorrect value given in AND, CHAR, CHR$, CLOSE, EOF, FOR, GOSUB, GOTO, HCHAR, INPUT, MOTION, NOT, OR, POS, PRINT, PRINT USING, REC, RESTORE, RPT$, SEG$, SIZE, VCHAR, or XOR.
- Array subscript value greater than 32767.
- File number greater than 255 or less than zero.
- More than three tones and one noise generator specified in SOUND.
- A value passed to a subprogram is not acceptable in the subprogram. For example, a sprite velocity value less than −128 or a character value greater than 143.
- Value in ON...GOTO or ON...GOSUB greater than the number of lines given.
- Incorrect position given after the AT clause in ACCEPT or DISPLAY.

67 **CAN'T CONTINUE**
- Program has been edited after being stopped by a breakpoint.
- Program was not stopped by a breakpoint.

69 **COMMAND ILLEGAL IN PROGRAM**
- BYE, CON, LIST, MERGE, NEW, NUM, OLD, RES, or SAVE used in a program.

ERRORS

84 **DATA ERROR**
- READ or RESTORE with data not present or with a string where a numeric value is expected.
- Line number after RESTORE is higher than the highest line number in the program.
- Error in object file in LOAD.

109 **FILE ERROR**
- Wrong type of data read with a READ statement.
- Attempt to use CLOSE, EOF, INPUT, OPEN, PRINT, PRINT USING, REC, or RESTORE with a file that does not exist or does not have the proper attributes.
- Not enough memory to use a file.

44 **FOR-NEXT NESTING**
- The FOR and NEXT statements of loops do not align properly.
- Missing NEXT statement.

130 **I/O ERROR**
- An error was detected in trying to execute CLOSE, DELETE, LOAD, MERGE, OLD, OPEN, RUN, or SAVE.
- Not enough memory to list a program.

16 **ILLEGAL AFTER SUBPROGRAM**
- Anything but END, REM, or SUB after a SUBEND.

36 **IMAGE ERROR**
- An error was detected in the use of DISPLAY USING, IMAGE, or PRINT USING.
- More than 10 (E-format) or 14 (numeric format) significant digits in the format string.
- IMAGE string is longer than 254 characters.

28 **IMPROPERLY USED NAME**
- An illegal variable name was used in CALL, DEF, or DIM.
- Using a TI Extended BASIC reserved word in LET.
- Using a subscripted variable or a string variable in a FOR.
- Using an array with the wrong number of dimensions.
- Using a variable name differently than originally assigned. A variable can be only an array, a numeric or string variable, or a user defined function name.
- Dimensioning an array twice.
- Putting a user defined function name on the left of the equals sign in an assignment statement.
- Using the same variable twice in the parameter list of a SUB statemtent.

ERRORS

81 **INCORRECT ARGUMENT LIST**
- CALL and SUB mismatch of arguments.

83 **INPUT ERROR**
- An error was detected in an INPUT.

60 **LINE NOT FOUND**
- Incorrect line number found in BREAK, GOSUB, GOTO, ON ERROR, RUN, or UNBREAK, or after THEN or ELSE.
- Line to be edited not found.

62 **LINE TOO LONG**
- Line too long to be entered into a program.

39 **MEMORY FULL**
- Program too large to execute one of the following: DEF, DELETE, DIM, GOSUB, LET, LOAD, ON...GOSUB, OPEN, or SUB.
- Program too large to add a new line, insert a line, replace a line, or evaluate an expression.

49 **MISSING SUBEND**
- SUBEND missing in a subprogram.

47 **MUST BE IN SUBPROGRAM**
- SUBEND or SUBEXIT not in a subprogram.

19 **NAME TOO LONG**
- More than 15 characters in variable or subprogram name.

43 **NEXT WITHOUT FOR**
- FOR statement missing, NEXT before FOR, incorrect FOR-NEXT nesting, or branching into a FOR-NEXT loop.

78 **NO PROGRAM PRESENT**
- No program present when issuing a LIST, RESEQUENCE, RESTORE, RUN, or SAVE command.

10 **NUMERIC OVERFLOW**
- A number too large or too small resulting from a *, +, −, / operation or in ACCEPT, ATN, COS, EXP, INPUT, INT, LOG, SIN, SQR, TAN, or VAL.
- A number outside the range −32768 to 32767 in PEEK or LOAD.

70 **ONLY LEGAL IN A PROGRAM**
- One of the following statements was used as a command: DEF, GOSUB, GOTO, IF, IMAGE, INPUT, ON BREAK, ON ERROR, ON...GOSUB, ON...GOTO, ON WARNING, OPTION BASE, RETURN, SUB, SUBEND, or SUBEXIT

ERRORS

25 **OPTION BASE ERROR**
- OPTION BASE executed more than once, or with a value other than 1 or zero.

97 **PROTECTION VIOLATION**
- Attempt to save, list, or edit a protected program.

48 **RECURSIVE SUBPROGRAM CALL**
- Subprogram calls itself, directly or indirectly.

51 **RETURN WITHOUT GOSUB**
- RETURN without a GOSUB or an error handled by the previous execution of an ON ERROR statement.

56 **SPEECH STRING TOO LONG**
- Speech string returned by SPGET is longer than 255 characters.

40 **STACK OVERFLOW**
- Too many sets of parentheses.
- Not enough memory to evaluate an expression or assign a value.

54 **STRING TRUNCATED**
- A string created by RPT$, concatenation ("&" operator), or a user defined function is longer than 255 characters.
- The length of a string expression in the VALIDATE clause is greater than 254 characters.

24 **STRING-NUMBER MISMATCH**
- A string was given where a number was expected or vice versa in a TI Extended BASIC supplied function or subprogram.
- Assigning a string value to a numeric value or vice versa.
- Attempting to concatenate ("&" operator) a number.
- Using a string as a subscript.

135 **SUBPROGRAM NOT FOUND**
- A subprogram called does not exist or an assembly language subprogram named in LINK has not been loaded.

14 **SYNTAX ERROR**

* An error such as a missing or extra comma or parenthesis, parameters in the wrong order, missing parameters, missing keyword, misspelled keyword, keyword in the wrong order, or the like was detected in a TI Extended BASIC command, statement, function, or subprogram.
* DATA or IMAGE not first and only statement on a line.
* Items after final ")".
* Missing "#" in SPRITE.
* Missing ENTER, tail comment symbol (!), or statement separator symbol (::).
* Missing THEN after IF.
* Missing TO after FOR.
* Nothing after CALL, SUB, FOR, THEN, or ELSE.
* Two E's in a numeric constant.
* Wrong parameter list in a TI Extended BASIC supplied subprogram.
* Going into or out of a subprogram with GOTO, GOSUB, ON ERROR, etc.
* Calling INIT without the Memory Expansion peripheral attached.
* Calling LINK or LOAD without first calling INIT.
* Using a constant where a variable is required.
* More than seven dimensions in an array.

17 **UNMATCHED QUOTES**

* Odd number of quotes in an input line.

20 **UNRECOGNIZED CHARACTER**

* An unrecognized character such as ? or % is not in a quoted string.
* A bad field in an object file accessed by LOAD.

ERRORS

Sorted by #

#	*Message*
10	NUMERIC OVERFLOW
14	SYNTAX ERROR
16	ILLEGAL AFTER SUBPROGRAM
17	UNMATCHED QUOTES
19	NAME TOO LONG
20	UNRECOGNIZED CHARACTER
24	STRING-NUMBER MISMATCH
25	OPTION BASE ERROR
28	IMPROPERLY USED NAME
36	IMAGE ERROR
39	MEMORY FULL
40	STACK OVERFLOW
43	NEXT WITHOUT FOR
44	FOR-NEXT NESTING
47	MUST BE IN SUBPROGRAM
48	RECURSIVE SUBPROGRAM CALL
49	MISSING SUBEND
51	RETURN WITHOUT GOSUB
54	STRING TRUNCATED
56	SPEECH STRING TOO LONG
57	BAD SUBSCRIPT
60	LINE NOT FOUND
61	BAD LINE NUMBER
62	LINE TOO LONG
67	CAN'T CONTINUE
69	COMMAND ILLEGAL IN PROGRAM
70	ONLY LEGAL IN A PROGRAM
74	BAD ARGUMENT
78	NO PROGRAM PRESENT
79	BAD VALUE
81	INCORRECT ARGUMENT LIST
83	INPUT ERROR
84	DATA ERROR
97	PROTECTION VIOLATION
109	FILE ERROR
130	I/O ERROR
135	SUBPROGRAM NOT FOUND

Index

The pages listed in *italics* show where the language elements are used in an illustrative program.

INDEX

INDEX

SERVICE AND WARRANTY INFORMATION

IN CASE OF DIFFICULTY

If TI Extended BASIC does not appear to be working properly, check the following:

1. Power — Be sure all devices are plugged in. Then turn on the power to the units in the proper sequence: Peripheral devices first (if you have them), followed by the console and monitor. Insert the TI Extended BASIC module carefully.
2. Connector Separation — Check for proper alignment of the console and any accessory devices such as the Disk Drive Controller, Speech Synthesizer, and RS232 Interface. Remove and reinsert the TI Extended BASIC module.
3. If none of the above procedures corrects the difficulty, consult "If You Have Questions or Need Assistance" or see the "Service Information" portion of the *User's Reference Guide* that came with your computer.

If you have questions concerning module repair or peripheral, accessory, or software purchase, please call our Consumer 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 questions about programming, specific applications, etc., you can call (806) 741-2663. Please note that this is not a toll-free number and collect calls cannot be accepted.

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

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."

THREE MONTH LIMITED WARRANTY

THIS TEXAS INSTRUMENTS TI EXTENDED BASIC COMMAND MODULE WARRANTY EXTENDS TO THE ORIGINAL CONSUMER PURCHASER OF THE MODULE.

WARRANTY DURATION: This command module is warranted to the original consumer purchaser for a period of three months from the original purchase date.

WARRANTY COVERAGE: This command module is warranted against defective materials or workmanship. **THIS WARRANTY IS VOID IF THE COMMAND MODULE HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE OR OTHER CAUSE NOT ARISING OUT OF DEFECTS IN MATERIAL 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 MODULE 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 module and determine the exchange policies of the retailer. During the above three month warranty period your TI Extended BASIC command module will be repaired or replaced with a new or reconditioned unit of the same or equivalent model (at TI's option) when the module is returned by prepaid shipment to a Texas Instruments Service Facility listed below. The repaired or replacement unit will be warranted for three months from the date of repair or replacement. Other than the postage requirement, no charge will be made for the repair or replacement of in-warranty modules. Texas Instruments strongly recommends that you insure the module for value prior to shipment.

Texas Instruments Consumer Service Facilities

U. S. Residents:
Texas Instruments Service Facility
2303 North University
Lubbock, Texas 79415

Canadian customers only:
Geophysical Services Incorporated
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
6700 Southwest 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503)643-6758

TEXAS INSTRUMENTS
INCORPORATED
DALLAS, TEXAS

Printed in U.S.A.

1041345-1

ADDENDUM

TI Extended BASIC Owner's Manual

The program listing on page 153 in the manual is incorrect. Line 110 should read:

```
>110 OPEN #1:"DSK1.RNDFILE",RELATIVE,INTERNAL
```


1053596-2
(Use with 1041345-1)

IMPORTANT PRODUCT INFORMATION FOR TI EXTENDED BASIC

TI Extended BASIC has been enhanced and modified for use with both the TI-99/4A and TI-99/4 Computers. Several important product differences should be noted in relation to the type of computer you have. Please read this folder and mark the appropriate changes in your copy of the *TI Extended BASIC* owner's manual.

Although the TI-99/4A and TI-99/4 Computers are similar, the TI-99/4A is easily recognizable by its standard typewriter keyboard which returns both upper-case (large capital) and lower-case (small capital) alphabetical characters. Depressing the **ALPHA LOCK** key locks the alphabet keys in upper-case mode. To release **ALPHA LOCK**, press the key again.

When the TI Extended BASIC module is in place, both the TI-99/4A and TI-99/4 Computers share several enhancements. However, each computer also has its own unique features. These features are discussed in the following paragraphs.

AUTO REPEAT FEATURE

When using TI Extended BASIC on either computer, holding down a key for more than one second automatically causes its symbol to be repeated on the display until you release the key.

SPECIAL FUNCTION KEYS

The TI-99/4A Computer has the same special computer functions as the TI-99/4. However, these functions are frequently assigned to different keys on the TI-99/4A Computer. The following chart compares the keystroke sequences for the function keys on the two units.

Function Keys

Key Name	*TI-99/4 Keys*	*TI-99/4A Keys*
AID	SHIFT A	FCTN 7
CLEAR	SHIFT C	FCTN 4
DELete	SHIFT F	FCTN 1
INSert	SHIFT G	FCTN 2
QUIT	SHIFT Q	FCTN =
REDO	SHIFT R	FCTN 8
ERASE	SHIFT T	FCTN 3
LEFT arrow	SHIFT S	FCTN S
RIGHT arrow	SHIFT D	FCTN D
DOWN arrow	SHIFT X	FCTN X
UP arrow	SHIFT E	FCTN E
PROC'D	SHIFT V	FCTN 6
BEGIN	SHIFT W	FCTN 5
BACK	SHIFT Z	FCTN 9
ENTER	ENTER	ENTER

In addition to these functions, the TI-99/4A Computer has functions represented as symbols on the fronts of the individual keyfaces. These functions may be accessed by pressing **FCTN** and the appropriate key simultaneously.

CONTROL KEYS

The TI-99/4A Computer also has control characters which are used primarily for telecommunications. To enter a control character, hold down the **CTRL** key and press the appropriate letter, number, or symbol key.

EXPANDED CHARACTER SET — TI-99/4A

As explained in your *TI Extended BASIC* manual, codes 32-95 are the predefined standard ASCII characters on the TI-99/4 Computer. The cursor and edge characters, ASCII codes 30 and 31, are assigned to character set 0. The undefined character codes (128-135 and 136-143) are assigned to sets 13 and 14, respectively.

These codes and the corresponding characters are listed in *Appendix C* of the manual. The CALL KEY character codes are also listed in *Appendix C*. *Appendix E* in the manual lists the 15 character code sets which may be used for color graphics.

Due to the inclusion of the lower-case character set, the defined characters on the TI-99/4A Computer are the standard ASCII characters for codes 32 through 127. The following chart lists these characters and their codes.

ASCII CODE	*CHARACTER*	*ASCII CODE*	*CHARACTER*
30	■ (cursor)	55	7
31	(edge character)	56	8
32	(space)	57	9
33	! (exclamation point)	58	: (colon)
34	" (quote)	59	; (semicolon)
35	# (number or pound sign)	60	< (less than)
36	$ (dollar)	61	= (equals)
37	% (percent)	62	> (greater than)
38	& (ampersand)	63	? (question mark)
39	' (apostrophe)	64	@ (at sign)
40	((open parenthesis)	65	A
41	) (close parenthesis)	66	B
42	* (asterisk)	67	C
43	+ (plus)	68	D
44	, (comma)	69	E
45	- (minus)	70	F
46	. (period)	71	G
47	/ (slant)	72	H
48	0	73	I
49	1	74	J
50	2	75	K
51	3	76	L
52	4	77	M
53	5	78	N
54	6	79	O

ASCII CODE	*CHARACTER*
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[(open bracket)
92	\ (reverse slant)
93	] (close bracket)
94	∧(exponentiation)
95	_ (line)
96	` (grave)
97	a
98	b
99	c
100	d
101	e
102	f
103	g

(97–103: Displayed on screen as small capitals.)

ASCII CODE	*CHARACTER*
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{ (left brace)
124	¦ (vertical line)
125	} (right brace)
126	~ (tilde)
127	DEL (appears on screen as a blank)

(104–122: Displayed on screen as small capitals.)

CALL KEY SUBPROGRAM

The information given on the KEY subprogram in Chapter 4 of the *TI Extended BASIC* manual is accurate for the TI-99/4 Computer. The values of 3, 4, and 5 are not accessible as key units.

However, the TI-99/4A maps key units 0 through 5 to specific modes of operation. If the *key-unit* is 0, the keyboard is mapped in whichever mode was specified by the previous CALL KEY program line.

If the *key-unit* is 1, input is taken from the left side of the keyboard. If the *key-unit* is 2, input is taken from the right side of the keyboard.

A *key-unit* of 3 maps the computer into the standard TI-99/4 keyboard mode. Both upper- and lower-case characters are returned as upper-case characters only. Function codes 1 through 15 are active, but no control characters are returned.

A *key-unit* of 4 places the computer in Pascal mode with both upper- and lower-case characters active. The function codes 129 through 143 and the control character codes 1 through 31 are also active.

The *key-unit* 5 maps the TI-99/4A Computer in the BASIC mode. Both upper- and lower-case characters are active. The active function codes are 1 through 15, and the active control character codes are 128 through 159 (and 187).

In addition, codes are assigned to the function and control keys so that these can be referenced by the CALL KEY subprogram in TI Extended BASIC. The codes assigned depend on the key-unit value specified in a CALL KEY program statement. The following tables show typical code assignments.

FUNCTION KEY CODES

Codes			
TI-99/4 & BASIC Modes	*Pascal Mode*	*Function Name*	*Function Key*
1	129	AID	FCTN 7
2	130	CLEAR	FCTN 4
3	131	DELete	FCTN 1
4	132	INSert	FCTN 2
5	133	QUIT	FCTN =
6	134	REDO	FCTN 8
7	135	ERASE	FCTN 3
8	136	LEFT arrow	FCTN S
9	137	RIGHT arrow	FCTN D
10	138	DOWN arrow	FCTN X
11	139	UP arrow	FCTN E
12	140	PROC'D	FCTN 6
13	141	ENTER	ENTER
14	142	BEGIN	FCTN 5
15	143	BACK	FCTN 9

CONTROL KEY CODES

Codes				
BASIC Mode	*Pascal Mode*	*Mnemonic Code*	*Press*	*Comments*
129	1	SOH	**CONTROL A**	Start of heading
130	2	STX	**CONTROL B**	Start of text
131	3	ETX	**CONTROL C**	End of text
132	4	EOT	**CONTROL D**	End of transmission
133	5	ENQ	**CONTROL E**	Enquiry
134	6	ACK	**CONTROL F**	Acknowledge
135	7	BEL	**CONTROL G**	Bell
136	8	BS	**CONTROL H**	Backspace
137	9	HT	**CONTROL I**	Horizontal tabulation
138	10	LF	**CONTROL J**	Line feed
139	11	VT	**CONTROL K**	Vertical tabulation
140	12	FF	**CONTROL L**	Form feed
141	13	CR	**CONTROL M**	Carriage return
142	14	SO	**CONTROL N**	Shift out
143	15	SI	**CONTROL O**	Shift in
144	16	DLE	**CONTROL P**	Data link escape
145	17	DC1	**CONTROL Q**	Device control 1 (X-ON)
146	18	DC2	**CONTROL R**	Device control 2
147	19	DC3	**CONTROL S**	Device control 3 (X-OFF)
148	20	DC4	**CONTROL T**	Device control 4
149	21	NAK	**CONTROL U**	Negative acknowledge
150	22	SYN	**CONTROL V**	Synchronous idle
151	23	ETB	**CONTROL W**	End of transmission block
152	24	CAN	**CONTROL X**	Cancel
153	25	EM	**CONTROL Y**	End of medium
154	26	SUB	**CONTROL Z**	Substitute
155	27	ESC	**CONTROL .**	Escape
156	28	FS	**CONTROL ;**	File separator
157	29	GS	**CONTROL =**	Group separator
158	30	RS	**CONTROL 8**	Record separator
159	31	US	**CONTROL 9**	Unit separator

You may also obtain detailed CALL KEY subprogram information, including keyboard diagrams, in your *User's Reference Guide* for the TI-99/4A Computer.

CALL VERSION SUBPROGRAM

The VERSION subprogram (discussed in Chapter 4 of your *TI Extended BASIC* manual) now returns a value of 110 on both computers.

DATA STATEMENT

The computer reads any information entered after a DATA statement as a part of the DATA statement. Therefore, in a multi-statement program line, a DATA statement should not be followed by another statement.

SCIENTIFIC NOTATION

Whenever you use scientific (or exponential) notation, be certain that the "E" is an upper-case (large capital) character. A lower-case "e" may cause your program to function improperly.

PRE-SCAN — !@P– and !@P+

After you enter RUN to start a program, you may notice a pause before the program actually begins. This pause is the time the computer takes to "pre-scan" your program to establish memory space for variables, arrays, and data. Then the computer proceeds through each instruction, performs the appropriate functions, and establishes variable values. Since the time required to pre-scan depends on the length of the program, you may want to decrease the pre-scan pause, particularly if you have a long program.

TI Extended BASIC's new pre-scan commands, !@P– and !@P+, allow you to control which instructions will not be pre-scanned. Because the purpose of the pre-scan is to set memory space for variables, only those instructions which contain the first reference to the variables need to be pre-scanned. Therefore, many other instructions in your program do not require a pre-scan.

Careful program planning is required to minimize the statements that need the pre-scan. When certain types of statements (as explained here) are used in your program, the procedures listed below should be included in the pre-scan.

- Enter your *first* DATA statement within the pre-scan.
- Include the first use of each variable and/or array. (Also, include the OPTION BASE statement, if used.)
- Include the first reference to each CALL statement of any subprogram.
- Include all DEF statements for user-defined functions.
- Include all SUB statements and SUBEND statements in the pre-scan.

Note that a variable in a user-defined (SUB) subprogram is considered to be unique from any other variable used elsewhere in your program, even though the name and value may be the same. Therefore, each variable used in a user-defined subprogram must be included in the pre-scan.

To use the pre-scan option, first be certain that your completed program runs successfully. Then, at the beginning of a group of function statements, use the !@P – command to "turn off" the pre-scan. The following statements will not be pre-scanned, allowing the execution of your program to begin more quickly. Any statements related to variable names (not previously referenced during pre-scan) return a syntax error if the pre-scan is "off." Note that !@P – cannot be followed by another statement in a multiple statement.

To resume the pre-scan, simply enter the command !@P + . This command causes the pre-scan to "turn on" and memory space for variables may be set. Remember to use the !@P + command before a SUB or SUBEND statement and do not incorporate this command as a part of a multiple statement.

You may choose to use the pre-scan feature several times throughout your program. By turning the pre-scan on and off, your program can begin to execute more efficiently. The effectiveness of the pre-scan is more noticeable in large programs than small programs. Note that when using the TI-99/4A Computer, the commands, !@P – and !@P +, may also be entered with a lower-case "p" character.

The following examples illustrate how to include the pre-scan statements in an existing program. The final example demonstrates the most efficient use of the pre-scan feature by making use of a GOTO statement.

Examples:

Original program:

```
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(42,"0F0F0F0F0F0F0F0F")
130 .
140 .
150 .
160 CALL HCHAR(12,17,42)
170 CALL VCHAR(14,17,96)
180 DELAY=0
190 FOR DELAY=1 TO 500
200 NEXT DELAY
210 DATA 3
220 .
230 .
```

With pre-scan control added:

```
 10 DATA 3
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(42,"0F0F0F0F0F0F0F0F")
125 !@P-
130 .
140 .
150 .
155 !@P+
160 CALL HCHAR(12,17,42)
170 CALL VCHAR(14,17,96)
180 DELAY=0
185 !@P-
190 FOR DELAY=1 TO 500
200 NEXT DELAY
210 .
220 .
230 .
```

Notice that the first DATA statement has been moved to the beginning of the program so that it is included in the pre-scan. By including statements 125, 155, and 185, the pre-scan is turned off and on and off again. This causes the program to begin to execute more quickly.

With GOTO added:

You have the added ability to "trick" the computer into establishing memory space for CALL statements, as well as variable-related statements, without actually performing those statements. To do this, simply use a GOTO instruction in your program. The following example demonstrates the original program adapted with a pre-scan and a GOTO statement.

```
 10 DATA 3
 20 GOTO 100::DELAY::CALL CHAR::CALL CLEAR::CALL HCHAR::CALL
    VCHAR::!@P-
100 CALL CLEAR
110 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(42,"0F0F0F0F0F0F0F0F")
130 .
140 .
150 .
160 CALL HCHAR(12,17,42)
170 CALL VCHAR(14,17,96)
190 FOR DELAY=1 TO 500
200 NEXT DELAY
210 .
220 .
230 .
```

Note that the GOTO method causes the necessary memory space to be reserved in line 20. However, the statements in line 20 do not execute until they are encountered further on in the program. Thus, as shown in the preceding and following examples, you can put all of your variable references together and your subprogram calls do not have to be syntactically correct. This can be the most efficient use of the pre-scan option.

```
100 GOTO 180::X,Y,ALPHA,BETA,Z=DELTA::DIM B(10,10)
110 CALL KEY::CALL HCHAR::CALL CLEAR::CALL MYSUB
120 DATA 1,3,STRING
130 DEF F(X)=1-X*SIN(X)
140 .
150 .
160 .
170 !@P-
180 .
190 .
200 .
```

PROGRAMMING WITH LOWER-CASE LETTERS

Device names must be entered in upper-case (large capital) letters only. For example, "DSK1" is a correct device name, but "Dsk1" is not. Any reference to a device name spelled in lower-case (small capital) letters results in an error message.

File names are also very specific. Not only are they exact as to the correct spelling, but they are also specific as to the use of upper- or lower-case letters. For example, the file name, MYFILE, is not the same file as Myfile (a combination of large and small capital letters). Any file name listed in part or whole by lower-case letters is not accessible by the TI-99/4 Computer. Only the TI-99/4A Computer can access a program named or called in lower-case letters.

Lower-case letters in DATA statements or quoted strings function correctly and offer a wide variety of programming techniques on the TI-99/4A Computer. However, lower-case quoted strings and data are not displayed if you run the program on a TI-99/4 Computer. If you plan to run your program on both the TI-99/4A and TI-99/4 Computers, take special care when using lower-case letters.

To display the lower-case letters in your TI-99/4A Computer program when the program is run on a TI-99/4 Computer, simply include the following statements. Small capital letters are created similar to those of the TI-99/4A. Be sure to allow adequate memory space and execution time.

```
100 FOR I=65 TO 90
110 CALL CHARPAT(I,A$)
120 B$="0000"&SEG$(A$,1,4)&SEG$(A$,7,4)&SEG$(A$,13,4)
130 CALL CHAR(I+32,B$)
140 NEXT I
```

Insertion of the above program lines into your TI-99/4A program allows pre-programmed lower-case characters to be displayed by the TI-99/4 computer.

SIZE COMMAND

The SIZE example, using the Memory Expansion unit discussed in Chapter 4 of your *TI Extended BASIC* manual, now informs you that you have 24488 "BYTES OF PROGRAM SPACE FREE".

TAIL REMARKS

If you previously programmed a TAIL REMark that is identical to the pre-scan instructions (!@P+ or !@P-), your program will no longer function properly. These groups of characters are now considered to be "reserved words" for the operation of the computer.

CORRECTION TO APPENDIX C

ASCII code 12 in *Appendix C* of your *TI Extended BASIC* manual should be stated as the "PROC'D" character rather than as the "CMD" character.

LARGE PROGRAM FILES

Some programs written with TI BASIC may be too large to run with TI Extended BASIC because TI Extended BASIC requires more system overhead than TI BASIC. If you attempt to load such a program, your system will lock up. Before you can continue, you must turn your computer off, wait several seconds, and then turn it on again.

Entering a CALL FILES(1) or CALL FILES(2) command before loading your program *may* free enough memory to run the program with TI Extended BASIC. (A full explanation of the CALL FILES command can be found in the Disk Memory System manual.)

If a CALL FILES command does not free enough memory, you must shorten your TI BASIC program by deleting statements until the program fits in the memory available with TI Extended BASIC. However, if you have a Memory Expansion unit, you can run the entire program by using the following procedure:

1. As a safety measure, make a backup copy of your TI BASIC program on a cassette tape or diskette.
2. With the Memory Expansion unit attached and turned on, load your program with TI BASIC. Next, delete several statements, and save the shortened program on cassette tape or diskette. Then try to load this shortened program with TI Extended BASIC.
3. Type the deleted statements back into the proper places in your program.
4. Save your program on a diskette only. You are now ready to run your program with TI Extended BASIC and the Memory Expansion unit.

Note: Programs converted in this fashion can only be run with TI Extended BASIC and with the Memory Expansion unit attached and turned on. They are not stored in PROGRAM format.

MEMORY EXPANSION UNIT AND CASSETTE-BASED PROGRAMS

The Memory Expansion unit adds 32K bytes of Random Access Memory (RAM) to the built-in memory of the computer. However, even with the Memory Expansion unit available, the largest TI Extended BASIC program that can be stored on a cassette tape is 12K bytes in size. Note that, although the length of the actual program is limited, utilizing the Memory Expansion unit provides other advantages. For example, with the unit attached and turned on, your program (which can be up to 12K bytes in length) is stored in the expansion RAM. The numeric data generated by the program is stored in the Memory Expansion unit and the string data is stored in the computer's built-in memory. Without the unit, the program must be shorter so that both it and the generated data can be stored in the computer's built-in memory.

CONTINUE COMMAND

A CONTINUE command is used to resume your program when you break by using a BREAK command or by pressing **CLEAR**. However, if your last command (before the CONTINUE command) results in a error, the program may not continue properly. Your final command to the computer before the CONTINUE command must be correct. If you receive an ERROR message, be sure to enter a correct command, such as a PRINT command, before resuming program execution.

MANUAL ERRORS

Page 39

The second sentence in the third paragraph of the "Numeric Constants" section should be corrected to read "...number is greater than 99 *or less than −99*, then ...."

Pages 79 and 150

The string used in a *string-expression* with the DISPLAY ... USING and PRINT ... USING statements may be more general than shown in the examples in the manual. For example, both of the following are valid statements.

```
PRINT USING A$:X,Y
DISPLAY USING RPT$("#",5)&V$:A(12)
```

Pages 89, 133, and 135

The GOSUB, ON GOSUB, and ON GOTO statements should not be used to transfer control to and from subprograms.

Page 114

If you press **CLEAR** when using the LIST command, the listing stops and cannot be restarted.

Pages 118 and 119

The graphic figures at the bottom of page 118 and the top of 119 should be reversed.

Page 185

The TAB function cannot be used in the PRINT ... USING or DISPLAY ... USING statements. Also, the second paragraph of this explanation should be corrected to read as follows: "If the number of characters already printed on the current record is less than *numeric-expression*, the next *print-item* is printed beginning on the position indicated by *numeric-expression*. If the number of characters already printed on the current record is greater than or equal to the position indicated by *numeric-expression*, the next *print-item* is printed on the next record beginning in the position indicated by *numeric-expression*.

Page 200

In Appendix H, Color Combinations, the color codes for the last two listings in the "Best" category should be as follows.

14, *10*	Magenta on Light Red
3, *16*	Medium Green on White

In the "Fourth Best" category, the third combination in the second column should read:

6, 2	Light Blue on Black.

Format Lines

The following list gives corrections that should be made to the indicated formats and also shows the present format information.

DIM Statement (page 76)

Correct Format: (*integer1*[,*integer2*]...[,*integer7*])[,...]

Present Format: (*integer1*[,*integer2*]...[,*integer7*][,...])

DISPLAY Statement (page 77)

Correct Format: [SIZE (*numeric-expression*)][:]*print-list*

Present Format: [SIZE (*numeric-expression*)][:]*variable-list*

DISPLAY ... USING Statement (page 79)

Correct Formats: USING *string-expression*[:*print-list*]
USING *line-number*[:*print-list*]

Present Formats: USING *string-expression*[:*variable-list*]
USING *line-number*[:*variable-list*]

LINPUT Statement (page 113)

Correct Format: #*file-number*[,REC *record-number*]:

Present Format: [[#*file-number*][,REC *record-number*]:]

PRINT ... USING Statement (page 150)

Correct Format: [#*file-number*[,REC *record-number*],]

Present Format: [#*file-number*[,REC *record-number*]]

SPRITE Subprogram (page 173)

Correct Format: *dot-column*[,*row-velocity*,*column-velocity*][,...])

Present Format: *dot-column*,[,*row-velocity*,*column-velocity*][,...])