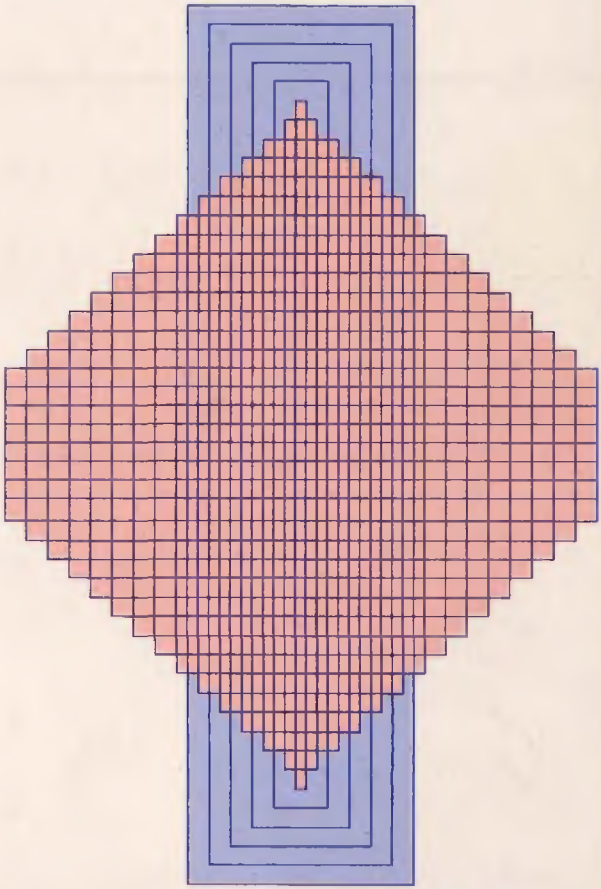
ATARI®

ST BASIC™

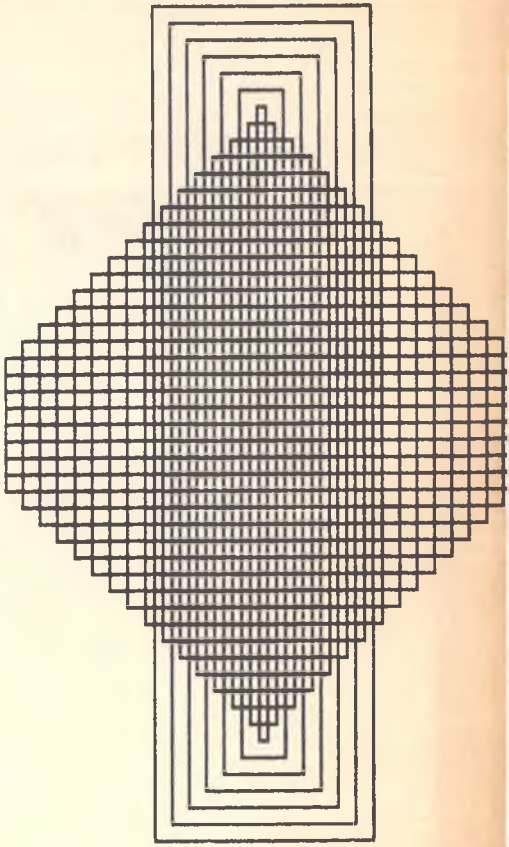
SOURCEBOOK AND TUTORIAL



A PROGRAMMING LANGUAGE FOR THE ST™ COMPUTER

ATARI®

ST BASIC™ Sourcebook and Tutorial



A Programming Language for the ST™ Computer

Every effort has been made to ensure the accuracy of the product documentation in this manual. However, because we are constantly improving and updating our computer software and hardware, Atari Corp. is unable to guarantee the accuracy of printed material after the date of publication and disclaims liability for changes, errors, and omissions.

ATARI, ST, ST BASIC, TOS, and 520ST are trademarks or registered trademarks of Atari Corp.

GEM is a trademark of Digital Research Inc.

VT is a trademark of Digital Equipment Corporation.

ATARI®

INTRODUCTION

BASIC is the most commonly used computer programming language. It is easy-to-learn, yet still a powerful programming tool. ST BASIC is very similar to the mainstream dialects of BASIC, but it takes advantage of the windows, drop-down menus, and graphic icons of the GEM™ Desktop. This version of BASIC also takes advantage of the speed and graphic capabilities of the ST™ Computer System.

How To Use This Manual

The *ST BASIC Sourcebook and Tutorial* is set up for easy access to all the information a programmer needs. Both the first-time programmer and the professional will find all the information to satisfy their level of programming expertise. It is recommended that you use this manual as a companion to your *ATARI ST Owner's Manual*.

Chapter 1, **Getting Started With ST BASIC**, is intended for use by all programmers. The chapter explains how to load the ST BASIC program and demonstrates the unique aspects of the program. For the experienced programmer, the section on "Writing an ST BASIC Program" is a short course on programming, showing how ST BASIC works within the GEM operating environment.

Chapter 2, **Programming With ST BASIC**, is a tutorial on ST BASIC showing the beginning programmer how to use and enjoy this unique version of the BASIC language.

Chapter 3, **ST BASIC Menus**, provides a detailed description of each ST BASIC menu. All menu options and Dialog Boxes that are accessible from the ST BASIC Desktop are individually described.

The Appendices to this manual provide a comprehensive reference source for every aspect of ST BASIC. Appendices A, B, C, D, and I are the primary references for most programmers. Appendix A lists every reserved word in the language. Appendix B details ST BASIC's operators, order of precedences, and reviews the syntax of the language. Appendix C defines and explains every ST BASIC command, function, and statement. Appendix D lists and describes each ST BASIC Error Message. And Appendix I, the **Glossary**, is

a convenient resource for all programmers. The remainder of the appendices offer specialized information for different levels of programming needs, including sample programs for beginning and advanced programmers.

Note: Advanced programmers can refer to the descriptions of GEMSYS and VDISYS in Appendix C of this manual for an introduction to the operating system interface.

The manual also has an **Index** to help you find information quickly and efficiently. And if you have any additional questions, refer to the **Customer Support** section.

For your convenience, a tear-out template showing the uses of the special function keys on the ST keyboard has been provided. Simply follow the instructions and attach the template to your computer.

TABLE OF CONTENTS

CHAPTER 1
GETTING STARTED WITH ST BASIC

This chapter provides a general introduction to ST BASIC and demonstrates how the language works within the desktop environment of the ST Computer System.

The chapter is divided into four main parts:

- Loading ST BASIC
- Touring the ST BASIC Desktop
- Writing an ST BASIC Program
- Enhancing ST BASIC's Memory

Note: Before you begin programming with ST BASIC, you should make a backup copy of the ST Language disk. Having a backup disk provides security against accidentally erasing or damaging your ST Language disk. Refer to the *ATARI ST Owner's Manual* for complete instructions on making a backup disk.

LOADING ST BASIC

To begin using ST BASIC, you need to load the language program into your ST Computer. Follow the instructions shown below to load ST BASIC. If you have a one-drive system, follow the instructions labeled, "With One Disk Drive." If you have a two-drive system, follow the instructions labeled, "With Two Disk Drives."

With One Disk Drive

1. With the ST Computer turned on and the GEM Desktop on the video display screen, double-click on the Floppy Disk B icon.

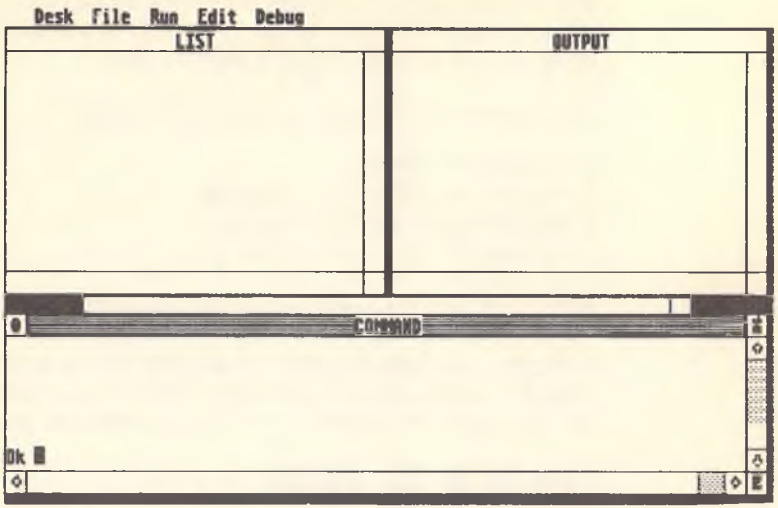
2. When the Dialog Box prompts you to insert Disk B into Drive A, place the ST Language disk into Drive A and press the **[Return]** key.

3. When the Floppy Disk window opens, double-click on the BASIC.PRG icon or filename. The ST BASIC Desktop will appear on the video display screen.

With Two Disk Drives

1. With the ST Computer turned on and the GEM Desktop on the video display screen, insert the ST Language disk into Drive B and double-click on the Floppy Disk B icon.

2. When the Floppy Disk B window opens, double-click on the BASIC.PRG icon or filename. The ST BASIC Desktop will appear on the video display screen.



The ST BASIC Desktop is the main point of reference for all your work with ST BASIC. The next two sections of this chapter show how to write a simple program in ST BASIC and how the ST BASIC Desktop works with the programming language.

Note: The ST BASIC screens in this manual are reproduced in high-resolution mode. If you are working in low-resolution, the screens will look differently on your display screen.

TOURING THE ST BASIC DESKTOP

ST BASIC uses the standard operating procedures of the GEM Desktop. The procedures for accessing menu items, selecting options, manipulating windows, and loading applications are explained in detail in the *ATARI ST Owner's Manual*.

Note: ST BASIC programs can be written in either upper- or lowercase characters. The program listings in this manual are presented in uppercase letters, but you can enter and run the programs in whichever case you choose.

Windows

The ST BASIC programming environment includes four windows: Command, Output, List, and Editor. After you load the ST BASIC program and the ST BASIC Desktop appears on the screen, the Command Window is active, and all four windows are available. (The Edit Window is available, but only a small part is visible under the List and Output Windows.)

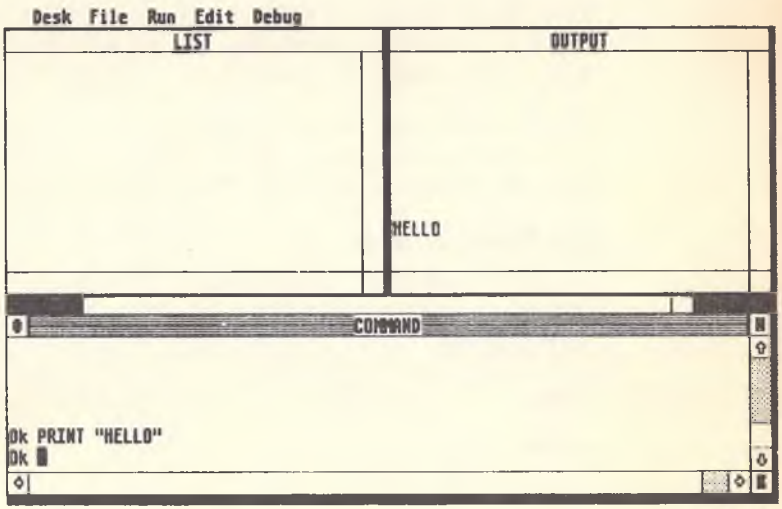
The procedures for sizing, moving, opening, closing, scrolling, and managing multiple windows are identical to the methods described in Chapter 4 of the *ATARI ST Owner's Manual*. Please refer to that manual for specific information.

The Command Window

ST BASIC commands and program lines are entered in the Command Window. The Ok prompt indicates that ST BASIC is ready for your command. Type:

```
PRINT "HELLO"
```

and press the **[Return]** key. The word HELLO will appear in the Output Window. Type your name and press **[Return]** to see how it works.



Note: If you type something ST BASIC doesn't understand, you will see the Error Message, "Something is wrong", in the Command Window. An up caret symbol(∧) will point to the place in the program statement where ST BASIC found an error. For a complete list of ST BASIC Error Messages, refer to Appendix D.

Your computer can function as a calculator by using the PRINT command. Type:

```
PRINT 2+2
```
[Return]

or use a question mark (?), the abbreviation for the PRINT statement. Type:

```
? 2+2
```
[Return]

The answer, 4, appears in the Output Window.

You can also use the numeric keypad for calculations. Type:

```
?
```
[Space]

then use the keypad to enter:

```
(5+3)*(6+2)/4+2
```
[Enter]

The answer, 18, is in the Output Window. Notice how ST BASIC handles arithmetic operations. The order of precedence is: Multiply, Divide, Add, Subtract. (Think of "My Dear Aunt Sally.")

Note: Whenever a word like **[Return]** or **[Esc]** is enclosed in square brackets in a programming example, press the corresponding key on the ST Computer keyboard.

The Output Window

ST BASIC uses the Output Window to display the results of commands or program operations. All program input prompts(?) and all output to the monitor appear in this window.

Type:

```
INPUT A
```

When you press **[Return]**, a question mark will appear in the Output Window. Type the number 2 and it appears in the Output Window. Now press **[Return]**. The Ok prompt will reappear in the Command Window.

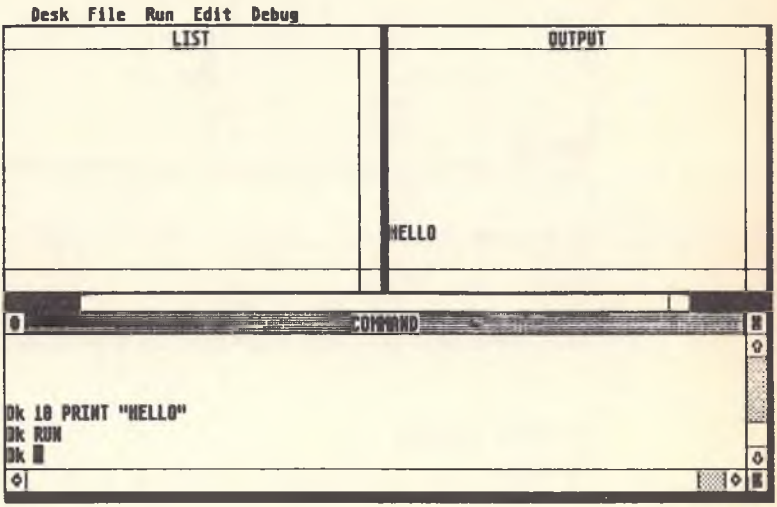
Type:

10 PRINT "HELLO" [Return]

You have just written a one-line program in ST BASIC! Type:

RUN [Return]

The word HELLO will appear in the Output Window.

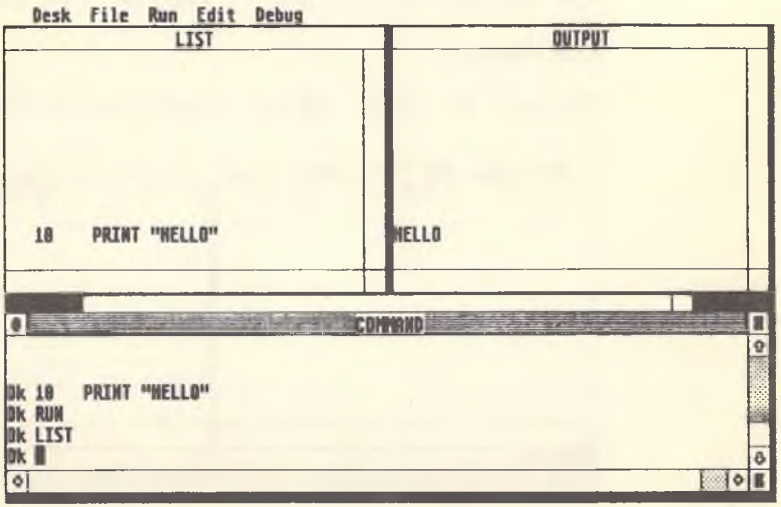


The List Window

Type:

LIST [Return]

Your one-line program will appear in the List Window. This window displays the program it has in memory. If you have a printer, you can print a listing of your program by typing LLIST.

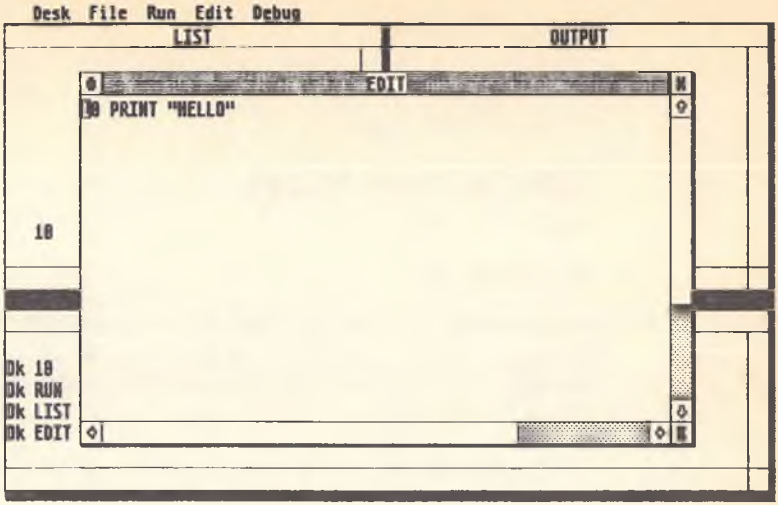


The Edit Window

Type:

EDIT [Return]

Your program will appear in the Edit Window. All editing is done in this window. Refer to "Writing an ST BASIC Program" for more information on the Edit Window. Press the [F10] key to leave the editor.



Menus

The Menu Bar is located along the top edge of the ST BASIC Desktop. The menu headings are Desk, File, Run, Edit, and Debug. Each heading has its own menu. To see the options within any menu, point at the menu heading with the mouse pointer. The menu will automatically drop down. If you don't want to select a menu item, click anywhere else on the ST BASIC Desktop. The menu will pop back up. Refer to Chapter 3 for a description of each menu.

Dialog Boxes and Error Messages

Dialog Boxes appear in the center of the ST BASIC Desktop whenever the program requires information that is not being provided in the program listing. Whenever an Error Message appears, information concerning an ST BASIC format or procedure will be displayed. For a complete listing of ST BASIC Error Messages, refer to Appendix D.

To exit from a Dialog Box, point at one of the Exit buttons (either the Ok or Exit button) and click on the left mouse button. If the Exit button has an enlarged border, you can press the **[Return]** key on the ST Computer keyboard rather than using the left mouse button.

Special Features

ST BASIC has three special features to make entering and reading your programs easier: AUTO Line Number Function, RENUM Function, and Labels.

AUTO Line Number Function

Type:

```
AUTO [Return]
```

Two asterisks and the number 10 will appear in the Command Window. The 10 is the first line number that generates the AUTO number function. The asterisks indicate that there is already a line 10 in memory.

Press [Return]. ST BASIC is now ready for you to enter line 20. You haven't entered a line 20 yet, so there aren't any asterisks.

Type:

```
PRINT "I'M YOUR FAITHFUL ATARI COMPUTER" [Return]
```

You now have a two-line program in memory. To stop the AUTO number function, press and hold down [Control], then press [G].

The Ok prompt will reappear in the Command Window. Type LIST to list your program. Since line 20 is too long for the List Window, click on the Size Box at the lower right edge of the List Window and stretch the window until it is long enough to incorporate the entire program listing.

RENUM Function

ST BASIC has a RENUM command that allows you to renumber your program automatically. RENUM uses your disk drive, so be sure you have a disk in it.

Note: This function will not work with a write-protected disk. To use the RENUM function, push the write-protect tab on the disk to the unprotected position. For more information on write protection, refer to Chapter 6 of the *ATARI ST Owner's Manual*.

Type:

```
RENUM 30,10,5 [Return]
```

When your disk drive stops and the Ok prompt reappears, list your program by typing LIST. The old line 10 has become line 30. The line number increment is 5, so the next line number is 35.

When you use the RENUM command, a copy of the renumbered program is automatically saved under the filename BASIC.WRK. The RENUM command is explained in detail in Appendix C.

Labels

ST BASIC lets you use symbolic labels to help identify program lines and subroutines. For example, using a statement like GOTO DONE instead of GOTO 300 makes for more readable listings and makes it easier to identify what each program line does for your program.

WRITING AN ST BASIC PROGRAM

This section shows you how to write and use simple programming techniques within the GEM Desktop environment. Follow the instructions carefully.

Note: When you load the ST BASIC program, the default (or boot-up) condition is lowercase letters for the ST BASIC programs. You can change this to uppercase by pressing the [Caps Lock] key.

Entering a Program

If there is a program listing in the List Window, clear it by typing:

CLEARW 1 [Return]

Then type:

NEW [Return]

This clears any current program from memory. Type:

LIST [Return]

The LIST Window will now be blank. Type:

AUTO [Return]

and enter the program on the following page. Notice that the line numbers are provided by ST BASIC. You do not have to enter the numbers.

```
10 REM COUNT.BAS
20 C=0
30 COUNT: ' INCREMENT THE VARIABLE C
40 C=C+1
50 PRINT C;
60 IF C=5 THEN PRINT "AGAIN!":GOTO 20
70 GOTO COUNT
```

Now you have the COUNT.BAS program in memory.

Press [Control] [G] to stop the AUTO function.

This simple program illustrates a few ST BASIC features.

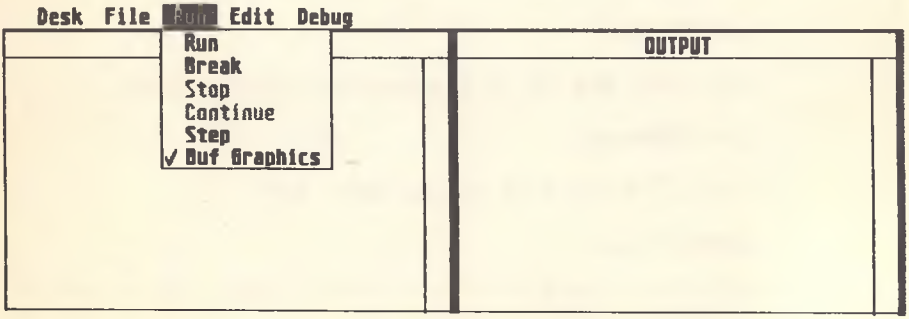
Line 10 has a REMark to help identify its function. The REMark is ignored by ST BASIC. You can begin REMarks with a single quote ('), as in line 30.

Line 30 is identified by the label COUNT; line 70 uses the same label in a GOTO statement. A label must be followed by a colon (:) when defined; it must not be an ST BASIC reserved word; it must begin with a letter; and it can't have any spaces in the label name.

Line 60 shows how to use the colon to put more than one command on a program line. You can put as many commands as you want on one line as long as you separate them with colons and the line is no longer than 249 characters. Multiple commands on a program line are referred to as a compound statement.

Running a Program

Select the Run menu from the Menu Bar.



Click on the Run option. You will see:

```
1 2 3 4 5 AGAIN!
```

printing continuously in the Output Window. To stop the program, click on the Break option in the Run menu. The message **— Break — at line . . .** tells you where the program stopped running. Type STOP **[Return]** to get out of Break mode. When your program is in Break mode you can still use programming commands.

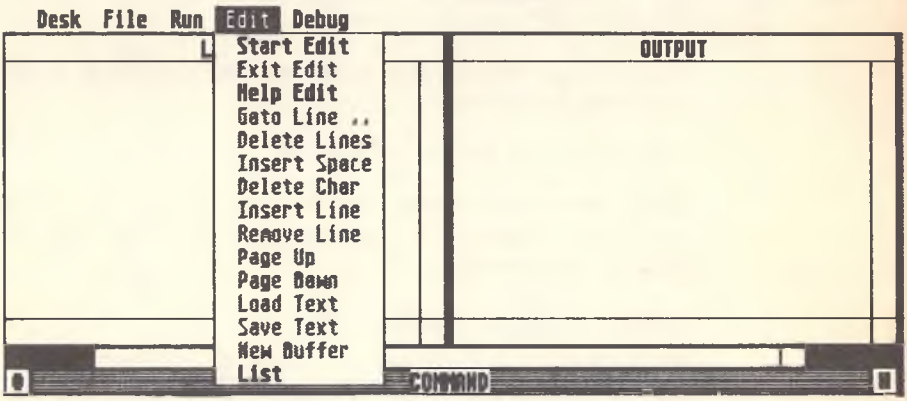
You can step (move) your program one line at a time by selecting the Step option from the Run menu. Each time you press **[Return]**, the program will be stepped forward. Notice that the program lines appear in the Command Window as they are executed. Type:

`END` [Return]

to cancel the STEP option.

Editing a Program

ST BASIC has an easy-to-use editor that allows you to make changes in your program without having to re-enter an entire program line. To edit your program, select the Edit menu.



Click the Start Edit option (or type ED).

When you edit, you use the cursor keys to move the cursor to the place on the screen where you want to insert or delete a character or add or delete space.

Use the cursor-controls keys to put the cursor on the first "A" in the word "AGAIN" in line 60. You can now type over the word "AGAIN". Type MORE. Notice that the type style changes to present a "ghost line". The "ghost lines" show you which lines have been edited but not put into the program memory. Press **[Return]**. You still need to discard the "N" in "MOREN." Line 60 changes back to the regular type style, indicating that the statement is again part of the program.

Select the Help Edit option in the Edit menu.

The Help Edit Dialog Box describes the function key commands available with ST BASIC.

Click on the Ok button to continue.

In the following example, use the function keys to edit the program. However, if you prefer, you can use the mouse and the Edit menu options.

Delete Char/Insert Space

With the cursor on the "N" in MOREN, press the [F2] key. The "N" disappears. Any time you press [F2], the character within the cursor is deleted and the text to the right of the cursor is moved one space to the left.

Move the cursor to the "M" in the word "MORE". Press [F1] 11 times. Type:

```
DO IT SOME
```

The line now reads:

```
60 IF C=5 THEN PRINT "DO IT SOME MORE!":GOTO 20
```

Press [Return] to enter the statement.

New Buffer

When you press [Return], the program lines you see in the Edit Window are put into the program memory buffer. To see what is actually in program memory, press [F9], New Buffer. The program memory is now duplicated in the Edit Window. If you haven't already pressed [Return], your original program will be in the Edit Window.

Insert Line/Delete Line

Move the cursor to line 30. Press [F4]. Line 30 is deleted from the program memory as indicated by the "ghost" type style. However, the line remains in the Edit Window until you press New Buffer [F9]. This feature makes it easy to correct your mistakes. Simply place the cursor on line 30 and press [Return]. Once you press New Buffer, the deleted line is erased from both the program memory and the Edit buffer. Press New Buffer. Line 30 is now erased.

Move the cursor to line 50 and press [F3], Insert Line. Now there is room to enter a new line. Type:

```
45 PRINT "COUNT";
```
[Return]

You can press New Buffer to see that the new line is in program memory.

The line numbers are beginning to get ragged, so renumber them.

Make room for a new line by pressing [F3]. Then type RENUM [Return]. When the cursor reappears, press New Buffer [F9] and the program is renumbered.

The program now has a mistake (a bug). Line 70 says GOTO COUNT, but you deleted the line labeled COUNT:.

Edit line 30 to read:

```
30 COUNT:C=C+1
```

You can RUN the program from the Edit Window by making room for a line and typing:

RUN [Return]

Press **[Control]** **[C]** to stop the program and return to the Edit Window.

Note: When a progam is requesting INPUT, use **[Control]** **[G]** to stop the program.

Load Text/Save Text

The ST BASIC editor will save the contents of the Edit Window to your disk. But the editor can only save 24 lines of text. If the program is longer than 24 text lines, none of the program lines outside the window will be saved.

Note: This function is different from the Save As function in the File menu. The Save As function saves complete programs which can be loaded and RUN. With Load Text/Save Text, you can't specify a filename, and the text saved doesn't necessarily have to be an ST BASIC program.

Press **[F8]**, Save Text. When the disk stops, your text has been saved.

Note: When you use the Save Text function, a file named BASIC.BUF is saved.

Make room for a blank line and type NEW **[Return]**. The NEW command clears the program area. Press New Buffer **[F9]**. The program area and Edit Windows are empty. To load the program back into the Edit Window, press **[F7]**. The program text is back! *Remember, the program memory is still empty!* Press New Buffer. The program display disappears. The program moves from the Edit Window to the program memory only when you press **[Return]** after each line.

Press **[F7]**, Load Text. Now press **[Return]** for each program line. Press New Buffer. Now your program is in both the Edit Window and the program memory.

Page Up/Page Down

The Page Up **[F5]** and Page Down **[F6]** functions allow you to edit programs that are larger than the program window. Page Up **[F5]** allows you to look at program lines toward the beginning of a program. Page Down **[F6]** takes you towards the end of a program.

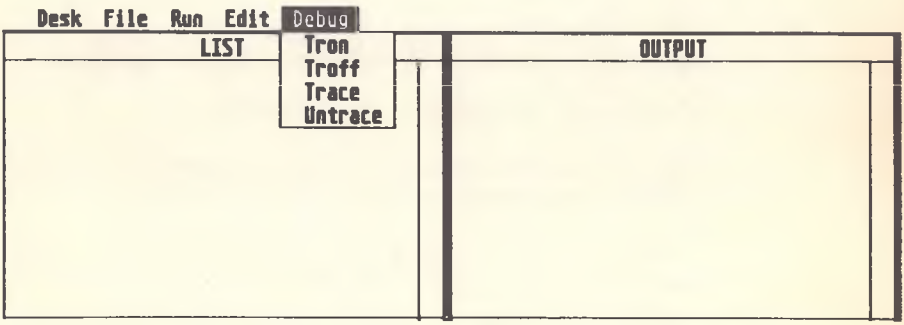
Note: The maximum visible line length is 80 characters in medium or high resolution. If you type off of the edge of the visible window, the text in the screen will move to the left so you can see what you're typing. You can type up to 80 characters in the Edit Window. If you attempt to edit a program that has lines longer than 80 characters, the part of the line beyond character 80 will be printed on the line below the first part of the line. It will only be included as part of the program line if the first character on the second line is a space. Otherwise, you must edit the line segments so that you can enter them as separately numbered program lines.

Leave the Editor by clicking the Exit Edit function or pressing [F10].

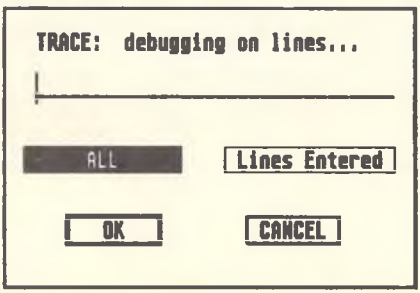
Debugging a Program

With the ST BASIC Debug menu, debugging a program is a simple process. Two options in the Debug menu help you see what a program is doing and what the problem might be. These options are Trace and Tron.

Select the Debug menu.



Click on the Trace option.

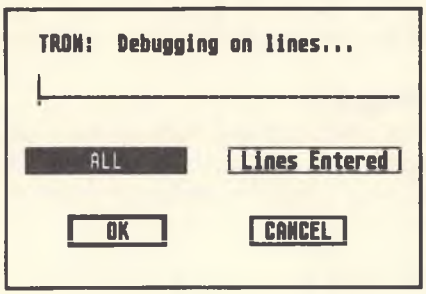


Click on the Ok button in the Dialog Box.

Now run your program. As each program line is executed, the Trace option displays the entire line in the Command Window.

To exit the Trace option, stop your program, select the Debug menu, and click on the Untrace option. Click on the OK button in the Trace Dialog Box.

Click on the Tron option in the Debug menu.



The Tron option displays the program's line number in the Command Window as each program line is executed.

Click on the Ok button in the Dialog Box.

Run your program again. As each program line is executed, Tron prints its line number in the Command Window.

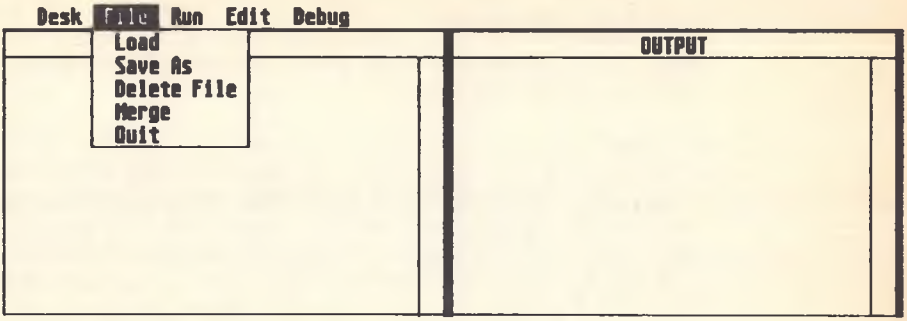
To exit the Tron option, stop your program, select the Debug menu, and click on the Troff option. Click the Ok button in the Dialog Box.

TRACE and TRON are explained in detail in Appendix C.

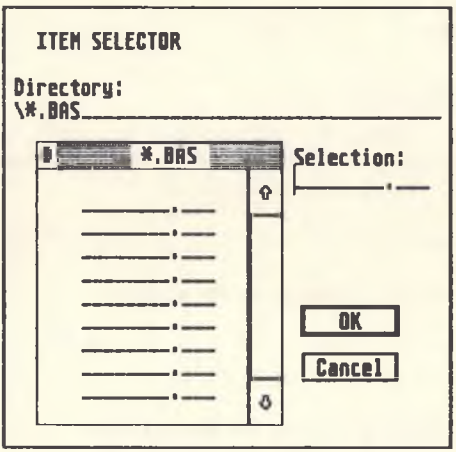
Note: The FOLLOW command is a debugging tool used in conjunction with TRACE and TRON. Refer to Appendix C.

Saving a Program

To save your program to disk, select the File menu from the Menu Bar.



Click on the Save As option.



Type COUNT in the Item Selector Dialog Box. Notice that ST BASIC appends a .BAS extender to your filename. The extender tells ST BASIC that the file is an ST BASIC program file. To store the file on the disk, click the OK button. When the Ok prompt appears, your file is saved.

You can also save the program by typing:

SAVE COUNT [Return]

ST BASIC will save the program as COUNT.BAS.

Note: The Save As option will replace (write over) an existing file with the same filename. If you type SAVE in the Command Window, you cannot save over a file that has the same filename.

Loading a Program

Type NEW [Return] to clear your program from memory. Then type LIST to insure that it's gone.

To load the program from disk, select the File menu and click on the Load option. COUNT.BAS will appear in the Item Selector Dialog Box. Select COUNT.BAS with the mouse pointer by clicking once on the program name, then click on the OK button. When the Ok prompt appears, your program will be in memory. To make sure, list it by typing LIST. The heading, "List of \COUNT.BAS", tells you the program is stored under the filename COUNT.BAS.

You can also load the program by typing:

```
LOAD COUNT
```

Merging Programs

Sometimes it's more convenient to write a program in small modules (parts) and assemble them at a later time. The Merge function allows you to do that.

Enter and save the following program as BOTTOM.BAS:

```
20 PRINT "MADE LONGER BY MERGING"
30 END
```

Type NEW and enter this program:

```
10 PRINT "THIS IS A SHORT PROGRAM"
20 END
```

Type RUN to run the program.

Select the Merge option from the File menu. Then select BOTTOM.BAS from the Item Selector Dialog Box and click on the Ok button.

List the program. As you can see, the two program segments are merged. Notice what happened to line 20. In the original program it was 20 END. The merged program's line 20 has replaced it. When you merge program segments, you need to plan your line numbers carefully.

Deleting a Program

To delete a program, select the File menu and click on the Delete File option. Click on the name of the file you want to delete. For example, click on BOTTOM.BAS. Then click on the Ok button. When the Ok prompt appears, the file has been deleted.

Leaving ST BASIC

To leave the ST BASIC programming environment, select the File menu and click on the Quit option. You can also leave ST BASIC by typing SYSTEM or QUIT.

Typing Commands

If you prefer, you can type programming commands from the ST keyboard instead of using the mouse. The typed commands are:

AUTO
[Control] [G] (To stop a program, to stop AUTO line numbering, or to place program into Break mode)
[Control] [C] (To stop and exit program without being able to continue)

CONT or [Return]
DELETE <*line number list*>
EDIT or ED (To enter edit)
ERA <*filename*> (To delete a file)
LOAD <*filename*>
MERGE <*filename*>
NEW
OLD <*filename*>
QUIT
REPLACE <*filename*>
RUN <*filename*>
SAVE <*filename*>
STEP
SYSTEM
TRACE
TROFF
TRON
UNTRACE

A complete list of ST BASIC reserved words is provided in Appendix A.

ENHANCING ST BASIC'S MEMORY

If you have a 520ST™ Computer System with TOS on a TOS System disk, use the instructions in this section to increase the available memory space for programming.

After you load TOS from the TOS System disk and ST BASIC from the ST Language disk, you will have a limited amount of memory space available for programming. This limitation is especially critical if you want to use the buffered graphics capabilities of the ST Computer.

If you have *any* ST Computer System with TOS in ROM, you can still increase the available memory space for programming by following the instructions under "Without Buffered Graphics" below.

With Buffered Graphics

If you want to use buffered graphics, 30,000 bytes of memory can be found by disabling the GEM desk accessories. There are two simple methods for disabling the desk accessories:

1. Delete the desk accessories from the TOS backup disk. Simply, open the TOS disk window and place the DESK.ACC file in the trash. Remember, you still have the file on the original language disk if you want to re-install and use the accessories.

2. Rename the desk accessory files. Select a DESK.ACC file, point at the File heading on the Menu Bar, and select the Show Info option. The Show Info Dialog Box displays a cursor at the end of the filename. Press the **[Backspace]** key on the keyboard until DESK.ACC is deleted. Rename the file to any name you wish as long as it does not have an .ACC extender.

Note: For detailed information on deleting and renaming files, refer to the *ATARI ST Owner's Manual*.

Without Buffered Grahics

If you don't want to use buffered graphics, you can add another 32,000 bytes of memory by turning off the Buffer Graphics option.

Point at the Run menu on the ST BASIC Desktop and see whether there is a check mark in front of Buf Graphics. If the check mark is present, select Buf Graphics. When the Dialog Box appears, click on the Ok button to turn off the Buffer Graphics option.

Note: If you turn off the Buffer Graphics option while you have a program in memory, the program will be lost from memory.

CHAPTER 2
PROGRAMMING WITH ST BASIC

When you write or use an ST BASIC program, you are instructing the computer to perform a task. ST BASIC is an interpretive language. You write the program in ST BASIC; ST BASIC then translates your instructions into machine code, the programming language the ST Computer understands.

COMMANDS, STATEMENTS, AND FUNCTIONS

The ST BASIC language uses three types of reserved (or key) words: commands, statements, and functions.

Each reserved word has a specific meaning in the ST BASIC language. It can only be used to express that meaning. You can't use a reserved word as a label, as a name for a program variable, or for any other purpose within a program.

Commands

A command tells the ST BASIC program to perform a function (i.e., load a program file or edit a program in memory) and lets you control ST BASIC while you are writing or debugging your program. However, you can't use a command within an ST BASIC program. For example, you can't use the LOAD command to load another program into memory while an ST BASIC program is running.

Statements

Statements tell the computer what function you want your program to perform. ST BASIC uses three types of statements: program statements, executable statements, and non-executable statements.

A program statement causes ST BASIC to do something while a program is running. The CHAIN statement, for example, instructs ST BASIC to bring another program into memory while the current program is running.

Executable statements (i.e., PRINT, INPUT, and GOTO) perform calculations. You can use executable statements within a program or enter them directly from the keyboard to the Command Window.

A statement made directly from the keyboard is a direct statement (made in "direct mode"). A statement executed within a program is an indirect statement (made in "indirect mode").

Non-executable statements aren't instructional. For example, the REM statement (short for REMark) tells ST BASIC to ignore the remainder of a program line rather than perform a new function. Non-executable statements are never used in the direct programming mode.

Functions

Functions assign values to variables. A variable is the name you assign to a value that changes as a result of calculations made during program execution. If you wanted to keep track of a total that changes its value as you add numbers to it in a program, you could assign the variable name TOTAL to that changeable value.

The function INT, for example, returns the whole number, or integer, value of a calculation. The program instruction, MYSHARE = INT (5/2), returns the number 2 because 2 goes into 5 twice with a fraction left over. The variable MYSHARE is assigned the value 2.

Functions have to be used in a complete ST BASIC sentence. For example, INT (5/2) by itself doesn't have any meaning. Used in a complete sentence, however, MYSHARE = INT (5/2) means "MYSHARE is equal to the whole number arrived at by dividing 5 by 2."

In ST BASIC, you declare a variable simply by using it with a function or statement. You can use any word as a variable name as long as it begins with a letter and isn't an ST BASIC reserved word (refer to Appendix A). MYSHARE, in the program example, is a variable because its value can be made to change, or vary, within a program.

You can use functions within programs or enter them directly from the keyboard.

INPUT AND OUTPUT (I/O)

Input is the information you enter into the computer. Output is the information the computer sends to you. Input is entered from a number of sources, including the keyboard, disk drive, modem, or MIDI port. Output is usually sent to the video monitor or a printer.

The INPUT statement takes input from the keyboard; the PRINT statement prints characters on the monitor; and the LPRINT statement prints characters on a printer.

Enter this program:

```
10 PRINT "EXCUSE ME..."
20 PRINT "WHAT'S YOUR NAME";
30 INPUT NAME$
40 PRINT "HELLO," ";NAME$;", IT'S NICE TO MEET YOU."
RUN [Return]
```

Notice the punctuation of the PRINT statement. Line 20 has a semicolon. Line 10 does not. When you run this sample program, the words between the quotation marks in line 10 are printed on one line followed by a line feed and a carriage return. Line 20 prints the words between the quotation marks because the semicolon stops the printer from going to the next line.

Notice also the question mark after "WHAT'S YOUR NAME". It's in the Output Window, but not in the program. INPUT prints the question mark in the Output Window as a prompt and waits for input.

INPUT has to refer to a variable. In the program, NAME$ is used as a string variable. A string variable is any series of characters that isn't interpreted as a number in the calculations. Putting a $ after a variable name designates it a string variable.

Line 40 shows the versatility of the PRINT statement. First, the string within the quotation marks is printed. Next, the string variable NAME$ prints on the same line (because of the semicolon). And then the last string within quotation marks is printed. ST BASIC places a carriage return and a line feed at the end of the program line because there isn't a semicolon after the last string.

The strings within the quotation marks are called string constants because they can't be modified by the INPUT statement.

The GOTOXY statement lets you print anywhere in the Output Window. The Output Window can hold 24 lines of text, each up to 80 characters long. If you consider the leftmost print column to be column 0 and the top line to be line 0, you can specify any print position in the window with two numbers. For example, 0,0 means column 0 in line 0; 5,8 means column 5 in line 8, and so on.

Enter this program:

```
10 ' MOVPRINT.BAS
20 FULLW 2: CLEARW 2: ' THIS CLEARS THE OUTPUT WINDOW
30 GOTOXY 0,0: PRINT "I'M GOING TO PRINT"
40 GOTOXY 5,5: PRINT "HERE"
50 GOTOXY 10,10: PRINT "...AND HERE"
60 GOTOXY 5,12: PRINT "...AND HERE!"
```

The previous two programs show how the statements, PRINT, INPUT, and GOTOXY are used. You'll learn variations of PRINT and INPUT in programming examples later in this chapter. For more information, look up the related reserved words in Appendix C.

DATA STATEMENTS

When you want to put different values into a variable, use a DATA statement.

Enter this program:

```
10 DATA 10,15,25,45
20 READ NUMBER
30 PRINT NUMBER
40 READ NUMBER
50 PRINT NUMBER
60 READ NUMBER
70 PRINT NUMBER
80 READ NUMBER
90 PRINT NUMBER
100 END
RUN
```

The program prints the numbers in the same order as they appear in the DATA statement. Each READ statement puts a DATA value into the variable NUMBER. Since NUMBER doesn't have a $ at the end, it's a numeric variable, not a string variable.

Remember that a variable is the name given to information that can change during a program's operation.

DATA statements can mix strings and numbers. Edit the previous program as shown and then run the program:

```
10 DATA ARTIE,15, NEIL,45
20 READ NAME$
30 PRINT NAME$,
40 READ SCORE
50 PRINT SCORE
60 READ NAME$
70 PRINT NAME$,
80 READ SCORE
90 PRINT SCORE
100 END
RUN
```

Notice how the comma in the PRINT line causes SCORE to tab.

If you have a printer, you can direct output to it from the program by changing PRINT to LPRINT.

The RESTORE statement lets you use a DATA list more than once in a program. Without the RESTORE statement, the program would use all the items in the DATA statement and the Error Message, "Out of data," would appear. Error Messages are computer messages that tell you something has gone wrong while the program is running.

Make the following change in line 100 in the program:

```
100 RESTORE 10
110 GOTO 20
```

The program will continue to print until you press [Control] [G].

PROGRAM CONTROL

Your computer will continue a calculation until you instruct it to stop. In the previous programming example, your computer could have read and printed an infinite number of names and scores. However, you would tire of typing the following data lines over and over again:

```
20 READ NAME$
30 PRINT NAME$,
40 READ SCORE
50 PRINT SCORE
```

You can use a programming technique called a loop to avoid tedious data input repetitions. The GOTO statement you added to the previous program is an example of a loop.

Loops

A loop is a program segment that repeats. Enter the following short program:

```
10 PRINT "HELLO"
20 GOTO 10
```

This is an example of an infinite loop with no exit condition. Most loops have built-in exit conditions so they can be controlled.

Press [Control] [G] to stop the loop.

FOR . . . NEXT Loops

The FOR . . . NEXT loop allows you to repeat a program segment a set number of times. Look at the following example:

```
10 FOR COUNT=1 to 4
20 READ NAME$, SCORE
30 PRINT NAME$, SCORE
40 NEXT COUNT
50 PRINT "DONE"
60 END
70 DATA BILL, 20, ADAM, 30, MICHAEL, 40, CHUCK, 45
```

The statements in lines 20 and 30 are the looping part of the program. Lines 10 and 40 set the loop conditions and boundaries.

When ST BASIC sees the FOR statement, it sets the variable COUNT to a starting value, in this case, 1. ST BASIC also notes that the maximum value of COUNT is 4.

After executing lines 20 and 30, ST BASIC encounters the NEXT statement in line 40. ST BASIC checks to see whether the value of COUNT is equal to or greater than the maximum given in the FOR statement. If COUNT is less than the maximum value, ST BASIC adds 1 to COUNT and goes to line 20.

When COUNT is equal to or greater than the given maximum, ST BASIC goes to the statement after NEXT.

FOR . . . NEXT loops are extremely versatile. You can use any numeric variable name in the FOR statement and any valid numbers to set the lower and upper limits of the loop. You can also tell ST BASIC to add a number other than 1 to the variable each time it passes through the loop. The following examples are legitimate FOR statements:

```
FOR DAY=5 TO 20

FOR SPEED=3 TO 7

FOR COUNT=2 TO 10 STEP 2
```

The last example tells ST BASIC to add a number other than 1 to the variable.

The following example tells ST BASIC to count backwards:

```
FOR COUNT=10 TO 0 STEP -1
```

Variable Limits

You can use a variable to set the loop limit. This lets you change the data in your program without changing the FOR statement. The following program shows a way to make the FOR . . . NEXT loop more flexible.

Edit the program as follows:

```
10 READ MAXIMUM
20 FOR COUNT=1 TO MAXIMUM
30 READ NAME$, SCORE
40 PRINT NAME$, SCORE
50 NEXT COUNT
60 PRINT "DONE"
70 END
80 DATA 5, BILL, 20, ADAM, 30, MICHAEL, 40, CHUCK, 45,
BARB, 50
```

The first value in the DATA statement, 5, is the number of name and score pairs in the DATA list. ST BASIC assigns the first value in the DATA statement to the variable MAXIMUM. Then it proceeds through the FOR . . . NEXT loop.

WHILE . . . WEND Loops

Enter this program:

```
10 READ NAME$, SCORE
20 WHILE NAME$ <> "STOP"
30 PRINT NAME$, SCORE
40 READ NAME$, SCORE
50 WEND
60 PRINT "DONE"
70 END
80 DATA BILL, 20, ADAM, 30, MICHAEL, 40, CHUCK, 45,
BARB, 50, STOP, 0
```

The program illustrates how a WHILE . . . WEND loop works. When ST BASIC sees the WHILE statement, it checks to see if the stated entry condition is true.

In the program, the entry condition is that NAME$ is not equal to STOP. (The symbols < and > mean "less than" and "greater than." Used together, they mean "not equal to.") As long as the entry condition is true, ST BASIC will execute the statements between WHILE and WEND. When the entry condition is no longer true and NAME$ is equal to STOP, ST BASIC goes to the next statement after WEND. Note that WHILE requires a value to check, so the program reads NAME$ and SCORE before entering the loop (at line 20) as well as within the loop (at line 40).

Note: An important difference between WHILE and FOR is that WHILE checks conditions before executing a loop and FOR checks conditions after executing it.

Conditional Loops

You can make your loop conditional by using the IF . . . THEN and GOTO statements.

IF . . . THEN statements are conditional statements. If a condition is met, then ST BASIC will perform a designated task.

Enter this program:

```
10 FOR COUNT=1 TO 5
20 IF COUNT=3 THEN PRINT "LOOK! A ";
30 PRINT COUNT
40 NEXT COUNT
50 END
```

Adding ELSE gives the program more flexibility. Edit the program as follows:

```
10 FOR COUNT=1 TO 5
20 IF COUNT=3 THEN PRINT "LOOK! A "; ELSE PRINT
"HO-HUM. A BORING ";
30 PRINT COUNT
40 NEXT COUNT
50 END
```

GOTO directs ST BASIC to a program line. Using IF . . . THEN and GOTO, you can make a loop.

Enter this program:

```
10 READ NAME$, SCORE
20 IF NAME$="STOP" THEN GOTO 60
30 PRINT NAME$, SCORE
40 READ NAME$, SCORE
50 GOTO 20
60 PRINT "DONE"
70 END
80 DATA BILL, 20, ADAM, 30, MICHAEL, 40, CHUCK, 45,
BARB, 50, STOP, 0
```

This program runs exactly like the program you wrote to demonstrate WHILE . . . WEND.

LOGICAL OPERATORS

Logic is the mechanics of valid decision making. Logical statements in programs allow you to reach valid conclusions from a set of premises. You can perform formal logic using the terms AND, OR, and NOT.

AND

AND tests for the condition that all of its terms are true. For example:

```
10 IF 2+2=4 AND 3+2=6 THEN PRINT "BRILLIANT!!" ELSE
PRINT "NOT BRILLIANT"
```

Run this program and it will print NOT BRILLIANT because one of the arguments, $3+2=6$, is not true. Change the argument to $3+2=5$, and the program will print BRILLIANT!!

You can string more than two arguments together with AND.

```
5 A$="GOODBYE"
10 IF 2+2=4 AND 3+2=5 AND A$="HELLO" THEN ?
"BRILLIANT!!" ELSE PRINT "NOT BRILLIANT"
```

This program will print NOT BRILLIANT. Change A$ to HELLO and the program will print BRILLIANT!!.

OR

OR tests for the condition that any of its terms are true. For example:

```
10 IF 2+2=4 OR 3+2=6 THEN PRINT "BRILLIANT!!" ELSE
PRINT "NOT BRILLIANT"
```

If you run this program, it will print BRILLIANT!! because one of the arguments, $2+2=4$, is true. Change the argument to $2+2=5$, and the program will print NOT BRILLIANT.

You can string more than two arguments together with OR. For example:

```
5 A$="GOODBYE"
10 IF 2+2=5 OR 3+3=5 OR A$="HELLO" THEN ?
"BRILLIANT!!" ELSE ? "NOT BRILLIANT"
```

This program will print NOT BRILLIANT. Change A$ to "HELLO" and it will print BRILLIANT!!.

NOT

NOT negates logical statements. For example:

```
5 A$="GOODBYE"
10 IF 2+2=5 OR 3+3=5 OR NOT A$="HELLO" THEN ?
"BRILLIANT!!" ELSE ? "NOT BRILLIANT"
```

Because it is not true that A$ = "HELLO", the statement NOT A$ = "HELLO" is true. The program prints BRILLIANT!! Look carefully at this program argument and run it to be certain that it is clear.

IMP

IMP is the abbreviation for implication. IMP checks the validity of conclusions drawn from premises.

IMP insures that you don't draw false conclusions from true premises. You can draw true conclusions from true premises, false conclusions from false premises, or true conclusions from false premises. For example:

```
10 IF 2+2=4 IMP 3+3=6 THEN PRINT "TRUE IMPLICATION"
ELSE ? "FALSE IMPLICATION"
20 IF 2+2=3 IMP 3+3=7 THEN PRINT "TRUE IMPLICATION"
ELSE ? "FALSE IMPLICATION"
30 IF 2+2=3 IMP 3+3=6 THEN PRINT "TRUE IMPLICATION"
ELSE ? "FALSE IMPLICATION"
40 IF 2+2=4 IMP 3+3=7 THEN PRINT "TRUE IMPLICATION"
ELSE ? "FALSE IMPLICATION"
```

This program prints:

```
TRUE IMPLICATION
TRUE IMPLICATION
TRUE IMPLICATION
FALSE IMPLICATION
```

ST BASIC can also perform logical operations on the bits of a byte. Bits, short for binary digits, are the logical building blocks used to represent letters and numbers in your computer. Each bit represents a 1 or a 0; eight bits make up a byte. A byte is a computer "word"—the amount of information it takes to express an integer from 1 to 255 or an alphabetic character.

Refer to Appendix B for an explanation of bitwise logic.

ON . . . GOTO

The ON . . . GOTO statement lets you run different parts of a program depending upon the value of a variable.

Enter this program:

```
10 CLEARW 2: REM PICK A PRIZE GAME
20 INPUT "ENTER A NUMBER BETWEEN 1 AND 5 TO SEE WHAT YOU
WIN!", PRIZE
30 ON PRIZE GOTO 50, 60, 70, 80, 90
40 PRINT "WATCH MY LIPS..." GOTO 10
50 PRINT "A HOUSE AND ";
60 PRINT "A CAR AND ";
70 PRINT "A VACATION AND ";
80 PRINT "A TV AND ";
90 PRINT "A STICK OF GUM"
100 INPUT "WANT TO TRY AGAIN (Y/N)"; ANSWER$
110 IF ANSWER$="Y" THEN GOTO 20
120 IF ANSWER$="N" THEN END
130 GOTO 100
```

Since there are five line numbers after the ON . . . GOTO statement, the numbers from 1 to 5 will send program execution to different lines. Any number outside that range lets ST BASIC go to the line after the ON . . . GOTO statement.

You can put the GOTO line numbers in any order and they will be selected in that listed order.

Change line 30 to:

```
30 ON PRIZE GOTO 90, 80, 70, 60, 50
```

This program example demonstrates a few new programming techniques.

Line 10 contains a remark statement. REM tells ST BASIC to ignore the rest of the line. REM lines let you put comments in a program to help clarify what the program is doing.

Lines 20 and 100 contain INPUT statements with assigned prompts. Remember, you can replace the ? prompt with your own prompt. These lines demonstrate two variations on the INPUT statement.

The comma after the prompt string in line 20 tells ST BASIC to print your prompt with no space or question mark after it.

In line 100, the semicolon tells ST BASIC to print a question mark and a space after your prompt.

These punctuation marks allow you to prompt with directions or questions within your program.

GOSUB . . . RETURN

The GOSUB . . . RETURN statement lets you use part of a program and then return to normal program flow without having to concern yourself with GOTO statements. Enter this program:

```
10 PRINT "THIS IS NORMAL PROGRAM FLOW."
20 GOSUB 60
30 PRINT "THIS IS THE REST OF NORMAL PROGRAM FLOW."
40 GOSUB 70
50 END
60 PRINT "THIS IS:"
70 PRINT "A SUBROUTINE."
80 RETURN
```

When ST BASIC sees the GOSUB statement, it goes to the specified program line. This is similar to GOTO, but ST BASIC remembers where the GOSUB statement occurred. When ST BASIC sees the RETURN statement, it returns to the statement immediately following GOSUB.

A program segment ending with RETURN is called a subroutine.

The previous example demonstrates that it isn't important where the GOSUB statement sends the ST BASIC program. As soon as ST BASIC encounters RETURN, it returns to the main program. Within the limits of available memory, you can have as many subroutines as you want in a ST BASIC program. However, each GOSUB must end with a RETURN statement.

ON . . . GOSUB . . . RETURN

The ON . . . GOSUB statement is identical to ON . . . GOTO except that it sends the ST BASIC program to a subroutine instead of to a different program line.

Examine the following program to see how ON . . . GOSUB . . . RETURN functions.

```
10 FOR I=1 TO 4
20 PRINT "THIS IS ";
30 ON I GOSUB 60, 70, 80, 90
40 NEXT I
50 END
60 PRINT "SUBROUTINE 1"
65 RETURN
70 PRINT "SUBROUTINE 2"
75 RETURN
80 PRINT "SUBROUTINE 3"
85 RETURN
90 PRINT "SUBROUTINE 4"
95 RETURN
```

PROGRAMMING TIPS AND SHORT CUTS

The Guessing Game

The Guessing Game program demonstrates some ST BASIC features that can make your programs more compact.

Enter this program:

```
10 REM A GUESSING GAME
20 ANSWER$="Y"
30 WHILE ANSWER$ <> "N"
40 COUNT=0: CLEARW 2: REM ***** CLEAR THE OUTPUT
WINDOW *****
45 REM ***** [:] ALLOWS MULTIPLE STATEMENTS ON
1 LINE *****
50 READ NUMBER: IF NUMBER=-1 THEN RESTORE: GOTO 50
60 PRINT: PRINT "I'M THINKING OF A NUMBER BETWEEN
1 AND 20.": PRINT
70 TRY: INPUT "YOUR GUESS: ", GUESS: ' ***** TRY IS A
LABEL *****
80 COUNT=COUNT+1: ' COUNT GUESSES
90 IF GUESS=NUMBER THEN GOTO OK: '***** OK IS A LABEL
USED WITH GOTO *****
100 IF GUESS > NUMBER THEN GOTO HIGH
110 ?: PRINT "NOPE. TOO LOW. TRY AGAIN.";: GOTO TRY: '
***** ? MEANS PRINT *****
```

```
120 HIGH: ?: ? "YOU GUESSED TOO HIGH. TRY AGAIN. ";:
GOTO TRY
130 OK: ?: ?" CONGRATULATIONS! YOU GOT IT IN "; COUNT;
" GUESSES!"
140 INPUT "DO YOU WANT TO PLAY AGAIN (Y/N)"; ANSWER$
150 IF ANSWER$ <> "Y" AND ANSWER$ <> "N" THEN GOTO 140
160 WEND
170 CLEARW 2: ? "THANKS FOR PLAYING!": END
180 ' ***** THE NUMBERS COME FROM THE NEXT LINE *****
190 DATA 15, 4, 19, 8, 10, 7, 17, 3, 14, 13, 5, 12, 1,
16, 11, 9, 12, 6, 18, -1
```

Look at the example carefully. You should understand how it works. Look up any unfamiliar ST BASIC words in Appendix C.

Labels

Instead of using line numbers in GOTO and GOSUB statements, you can specify a label (LABEL:) for a particular line and refer to the line by its name. This makes programs easier to read. When you look at a program months after it has been written, the labels will help you understand what you did.

REMarks (')

REMarks also help to make a program easier to read. Putting comments in your work will help you to remember the purpose of a particular routine. The apostrophe (') is a convenient substitute symbol for REM. REMarks take up memory space, so make them brief. All text entered in a statement after the REM (') will not be executed as part of the statement.

PRINT (?)

You can use the question mark instead of the word PRINT in an ST BASIC program. The question mark symbol (?) is easy to type and quickens program entry.

Multiple Line Statements (:)

Entering multiple statements after a single line number makes short subroutines easier to read. Be sure to separate the statements with colons (:).

Note: Statements which follow an IF . . . THEN statement will only be executed after the conditional IF .. . THEN is satisfied.

NEXT

FOR...NEXT is generally expressed as:

```
FOR I=1 TO 10: NEXT I
```

You can also express FOR . . . NEXT as:

```
FOR I=1 TO 10: NEXT
```

You can leave off the variable reference after NEXT. (After NEXT the previous FOR will be executed.) When you have a nested loop—a loop running within a loop—be certain that each FOR has a corresponding NEXT statement. Although you do not have to have a variable reference after each NEXT, within nested loops it is sometimes advisable to keep the variable references as they allow you to trace the loop when you edit or debug the program.

END

The END statement should be the last statement executed in an ST BASIC program. It ensures that files and variables are left in an orderly state. END is an optional statement, but leaving a program in a disorderly state can cause unexpected problems during program debugging.

GRAPHICS

Before you begin drawing graphs and pictures, you need to understand how the display screen functions.

Screen Resolution

The Output Window is divided into elements called pixels. Pixels are the spots on the screen that ST BASIC uses to plot lines and draw circles. Imagine the window to be a sheet of grid paper. Each box corresponds to a pixel. Pixels are counted beginning at the top left corner. The number of pixels available when the window is at full size depends upon the screen resolution you have selected.

Low Resolution

The window is 303 pixels wide by 166 pixels high.

Medium Resolution

The window is 607 pixels wide by 166 pixels high.

High Resolution

The window is 615 pixels wide by 344 pixels high.

ST BASIC can also draw in colors. The number of colors available also depends upon the screen resolution you use.

Low Resolution

16 colors.

Medium Resolution

4 colors.

High Resolution

Black and white.

Graphics Statements

The ST BASIC graphics statements are:

CLEARW
FULLW
COLOR
FILL
CIRCLE
PCIRCLE
ELLIPSE
PELLIPSE
LINEF

CLEARW

In the previous program example, CLEARW 2 cleared the Output Window. CLEARW can also clear the Command, List, and Edit Windows.

CLEARW 0 Clears the Edit Window.

CLEARW 1 Clears the List Window.

CLEARW 2 Clears the Output Window.

CLEARW 3 Clears the Command Window.

FULLW

FULLW makes a window full-screen size. The screen numbers are the same as for CLEARW:

FULLW 0 Makes the Edit Window full size.

FULLW 1 Makes the List Window full size.

FULLW 2 Makes the Output Window full size.

FULLW 3 Makes the Command Window full size.

COLOR

The COLOR command controls three parameters:

1. The color of the text printed to the screen.

2. The FILL color used for filling shapes and window background.

3. The color used for lines drawn in the Output Window.

Note: In the following programs the screen is usually set for medium resolution. Set the resolution to low and see how the program looks.

The following program illustrates the COLOR command:

```
10 COLOR 1,0,1
20 PRINT "BLACK LETTERS"
30 COLOR 2,0,1
40 PRINT "RED LETTERS"
50 COLOR 3,0,1
60 PRINT "GREEN LETTERS"
70 FOR I=1 TO 5000: NEXT
75 COLOR 1,0,1
80 END
```

The first number changes the color of text. The color stays the same until the next COLOR command. Edit the program as follows:

```
10 COLOR 0,0,1
20 PRINT "WHITE LETTERS"
30 COLOR 2,0,1
40 PRINT "RED LETTERS"
50 COLOR 3,0,1
60 PRINT "GREEN LETTERS"
70 FOR I=1 TO 5000: NEXT
75 COLOR 1,0,1
80 END
```

The second number changes the FILL color used for background color and PCIRCLE and PELLIPSE statements. The third number sets the color for drawing pictures and graphs.

Enter this program:

```
10 REM COLORS.BAS
20 WHITE=0: BLACK=1: RED=2: GREEN=3
30 COLOR BLACK,RED,GREEN
40 FULLW 2: CLEARW 2: FILL 0,0:' **** SETS BACKGROUND
TO FILL COLOR (RED)
50 PRINT "BLACK TEXT, RED FILL, GREEN CIRCLE, ";
60 CIRCLE 100,80,50
70 GOSUB DELAY
80 COLOR BLACK,WHITE,GREEN: ' ***** MAKE FILL WHITE
90 PCIRCLE 100,80,50,0,450: ' ***** WE'LL EXPLAIN
PCIRCLE LATER
100 PRINT "WHITE WEDGE";
110 GOSUB DELAY: GOSUB DELAY
120 END
130 DELAY: FOR I=1 TO 2500: NEXT: RETURN
```

The following chart shows the numbers for colors in different screen resolutions:

COLOR RESOLUTION

Color	Number	Low	Med	High
WHITE	0	X	X	X
BLACK	1	X	X	X
RED	2	X	X	
GREEN	3	X	X	
BLUE	4	X		
DARK BLUE	5	X		
BROWN	6	X		
DARK GREEN	7	X		
GREY	8	X		
DARK GREY	9	X		
LIGHT BLUE	10	X		
BLUE GREEN	11	X		
LIGHT PURPLE	12	X		
DARK PURPLE	13	X		
LIGHT YELLOW	14	X		
DARK YELLOW	15	X		

Note: The colors listed in the chart are the default colors, the colors in the palette when you turn on the computer. They can be changed with the Control Panel or by writing an ST BASIC program that specifically changes the colors.

The COLOR command can also select patterns to replace the solid fill color. Enter and RUN this program:

```
5 ' PATTERNS.BAS
10 COLOR 1,3,1,4,4: ' ***** FILL SCREEN WITH GREEN
FUJIS
20 FULLW 2: CLEARW 2: FILL 0,0
30 FOR P=2 TO 4
40 FOR I=1 TO 5
45 GOTOXY 8,13: ? " COLOR 1,2,1,;I;I;P
50 COLOR 1,2,1,I,P: ' I AND P CHOOSE PATTERN. FILL IS
RED
60 PCIRCLE 150,80,80: ' ***** PCIRCLE USES FILL COLOR
70 FOR D=1 TO 700:NEXT D: ' ***** A DELAY LOOP
80 NEXT I
90 NEXT P
95 COLOR 1,0,1,1,1: ' ***** RETURN TO DEFAULT COLORS
100 END
```

FILL

The FILL command allows you to fill a shape with color or a pattern. Enter and RUN this program:

```
5 ' PATTERN5.BAS
10 COLOR 1,3,1,1,1: ' ***** GREEN WINDOW
20 FULLW 2: CLEARW 2: FILL 0,0
30 COLOR 1,2,1,1,1: ' ***** SOLID RED FILL
40 CIRCLE 150,80,80
50 FILL 150,80
60 FOR D=1 TO 700: NEXT
70 COLOR 1,1,1,4,4: ' ***** BLACK PATTERN FILL
80 FILL 150,80
90 END
```

The FILL command is explained in detail in Appendix C.

CIRCLE and PCIRCLE

Drawing the outlines of circles and arcs is easy with ST BASIC. Simply locate the center of a circle and decide on its radius. The location of the center is specified in pixels. So is the radius. If you want the center 100 pixels to the right and 80 pixels down from the top left corner, you would say the center is located at 100,80. To draw a circle at that location, enter the following program:

```
10 COLOR 1,0,1,1,1: FULLW 2: CLEARW 2
20 CIRCLE 100,80,50
30 FOR I=1 TO 5000: NEXT
40 END
```

The first two parameters of the CIRCLE command give the horizontal and vertical position of the center. The third parameter is the radius. CIRCLE 100,80,50 places a circle at 100 pixels to the right and 80 pixels down from the top left corner, then draws the circle with a radius of 50 pixels.

The exact size and position of your circle depend upon the screen resolution you select. Most of the examples look best in medium resolution. The following program, which makes circles in all the colors available, is written to be run in low resolution. Run it in medium resolution to see the variation.

```
10 COLOR 1,0,1: FULLW 2: CLEARW 2: R=1: ' ***** RADIUS
IS 1 *****
20 FOR C=0 TO 15: ' ***** 16 COLORS *****
30 COLOR 1,0,C
40 FOR I=1 TO 5: ' ***** 5 CIRCLES PER COLOR *****
50 CIRCLE 150, 80, R
60 R=R+1
70 NEXT
80 NEXT
90 FOR I=1 TO 5000: NEXT: ' ***** PAUSE *****
100 END
```

You can use circles to make interesting patterns. The following program example shows how to draw multiple circles using different centers and changing radii:

```
10 COLOR 1,0,1: FULLW 2: CLEARW 2
20 PRINT "  MORE PATTERNS";
30 V=80: R=100
40 FOR H=200 TO 400 STEP 2
50 CIRCLE H, V, R
60 NEXT
70 H=300
80 FOR R=1 TO 98 STEP 2
90 CIRCLE H, V, R
100 NEXT
110 FOR I=1 TO 5000: NEXT: END
```

The CIRCLE command can also draw arcs. Enter this program:

```
5 REM SEMICIRC.BAS
10 COLOR 1,0,1: FULLW 2: CLEARW 2
20 FOR I=20 TO 200 STEP 20
30 CIRCLE 300,100,I,0,1800
40 NEXT
50 FOR I=1 TO 5000: NEXT
60 END
```

The CIRCLE command's last two parameters specify the starting and ending angles of an arc. Zero degrees is to the right of the Output Window; 180 degrees is to the left. Ninety degrees is at the top. Notice that the angles are in tenths of a degree—the 180 degrees is written as 1800.

Note: PCIRCLE is similar to CIRCLE except that it draws solid figures and segments rather than outlines of figures.

ELLIPSE and PELLIPSE

The ELLIPSE command is similar to CIRCLE except that an ellipse has both a horizontal and vertical radius.

ELLIPSE 200,80,80,20 draws a horizontal ellipse.

ELLIPSE 200,80,20,80 draws a vertical one.

The ELLIPSE command can also draw arcs. Enter this program:

```
5 ' SEMELIPS.BAS
10 COLOR 1,0,1: FULLW 2: CLEARW 2
20 FOR I=10 TO 100 STEP 10
30 ELLIPSE 300,100,I,50,0,1800
40 NEXT
50 END
```

Just as with CIRCLE, the last two parameters are the starting and ending angles in tenths of a degree.

Note: PELLIPSE is similar to ELLIPSE except that it draws solid figures instead of outlines of figures. The figures are drawn in the color specified by the second parameter of the previous COLOR statement.

Edit the previous program as shown:

```
5 ' SPELLIPS.BAS
10 COLOR 1,1,1: FULLW 2: CLEARW 2
20 FOR I=10 TO 100 STEP 10
30 PELLIPSE 300,100,I,50,0,1800
40 NEXT: COLOR 1,0,1
50 END
```

LINEF

The LINEF command draws lines. To draw a line, you specify the beginning and ending points. LINEF 50,50,100,100 draws a line between the points located at 50,50 and 100,100. Enter this program:

```
5 FULLW 2
10 CLEARW 2
20 LINEF 50,50,100,100
30 FOR I=1 TO 5000: NEXT: END
```

The following program displays a parade of lines in different colors:

```
10 C=0: TOP=40: BOTTOM=120
20 FOR H=20 TO 580 STEP 20
30 COLOR 1,0,C
40 LINEF H,TOP,H,BOTTOM
50 IF C<4 THEN C=C+1 ELSE C=1
60 NEXT
70 FOR I=1 TO 5000: NEXT: END
```

The following program uses color, text, and lines to graph monthly sales:

```
10 COLOR 1,0,1: FULLW 2: CLEARW 2
15 ' ***** FOUR SPACES AT THE BEGINNING, ONE AT
THE END *****
20 ? "   JAN FEB MAR APR MAY JUN JUL AUG SEP OCT
NOV DEC";
30 LINEF 27,92,453,92
40 FOR C=32 TO 384 STEP 32: ' ***** C IS COLUMN
COUNTER *****
50 READ SALES: AVERAGE = AVERAGE+SALES
60 IF SALES<60 THEN COLOR 1,1,2 ELSE COLOR 1,1,3
70 TOP=SALES: GOSUB BAR
80 NEXT
90 AVERAGE=AVERAGE/12
100 IF AVERAGE<60 THEN COLOR 2,1,2 ELSE COLOR 3,1,3
110 ? "AVG";
120 TOP=AVERAGE: C=424: GOSUB BAR
130 FOR D=1 TO 5000: NEXT
140 COLOR 1,0,1: END
150 DATA 50, 70, 60, 56, 50, 45, 70, 90, 100, 80, 70, 59
160 BAR: FOR P=1 TO 24: ' ***** DRAWS VERTICAL
BAR *****
165 ' ***** SUBTRACT VALUE FROM BOTTOM OF GRAPH TO
MAKE IT RIGHT SIDE UP ****
170 LINEF C+P, 152-TOP,C+P,152
180 NEXT
190 RETURN
```

You can use the LINEF command to draw almost any shape. By supplying a series of points, you can make LINEF form shapes by "connecting the dots." You can assemble different shapes into pictures and vary the location of a shape series. The following program shows how to draw a house wherever you want to put it in the Output Window. By adding the horizontal offset (HO) and vertical offset (HO) values, you can move the picture around the screen.

```
10 ' ***** DRAW A PICTURE *****
15 CLEARW 2
20 INPUT "POSITION (X,Y)"; H0, V0
30 COLOR 1,0,1
40 READ COUNT
50 FOR PART=1 TO COUNT
60 GOSUB DRAW
70 NEXT
80 FOR I=1 TO 5000: NEXT: END
90 DRAW: ' ***** DRAW A SHAPE *****
100 READ H1,V1,H2,V2
110 WHILE H2 <> 999
120 LINEF H0+H1,V0-V1,H0+H2,V0-V2
130 H1=H2: V1=V2
140 READ H2,V2
150 WEND
160 RETURN
170 DATA 5
180 ' ***** FRAME *****
190 DATA 0, 12, 0, 0, 40, 0, 40, 12, 999, 0
200 ' ***** ROOF *****
210 DATA -4, 10, 20, 27, 44, 10, 999, 0
220 ' ***** CHIMNEY *****
230 DATA 0, 12, 0, 22, 4, 22, 4, 15, 999, 0
240 ' ***** WINDOW *****
250 DATA 4, 3, 4, 9, 20, 9, 20, 3, 4, 3, 999, 0
260 ' ***** DOOR *****
270 DATA 26, 0, 26, 9, 36, 9, 36, 0, 999, 0
```

SOUND

The SOUND command lets you create tones or play music with your ST Computer.

The SOUND command controls three musical voices and a noise channel. You can select a voice and control the voice's volume, the notes played, the note's octave, and the note's duration.

To write music, you should understand octaves and the relationship of the notes in a scale to your ST Computer's terms. The notes C, C#, D, D#, E, F, F#, G, G#, A, A#, B. are read as 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 by the computer. Your computer numbers the octaves 1 through 8. As a point of reference, the 440Hz note A is note 10 of octave 4.

The following program example plays a song that's a favorite among computers:

```
10 VOICE=1: VOLUME=8
20 READ VOLUME,NOTE,OCTAVE,DURATION
30 WHILE NOTE <> 0
40 SOUND VOICE,VOLUME,NOTE,OCTAVE,DURATION
50 SOUND 1,0: ' ***** TURNS VOICE 1 OFF *****
60 READ VOLUME,NOTE,OCTAVE,DURATION
70 WEND
80 END
90 DATA 8,12,4,25, 8,9,4,25, 8,5,4,25, 8,12,3,25,
8,2,4,5
100 DATA 8,4,4,5, 8,5,4,5, 8,2,4,15, 8,5,4,5,
8,12,3,30, 0,1,4,15
110 DATA 8,7,4,25, 8,12,4,25, 8,9,4,25, 8,5,4,25,
8,2,4,5
120 DATA 8,4,4,5, 8,5,4,5, 8,7,4,15, 8,9,4,5,
8,7,4,30, 0,1,4,15
130 DATA 8,9,4,5, 8,10,4,5, 8,9,4,5, 8,7,4,5,
8,12,4,15, 8,9,4,5
140 DATA 8,7,4,15, 8,5,4,5, 0,1,4,15
150 DATA 8,7,4,5, 8,9,4,15, 8,5,4,5, 8,2,4,15,
8,5,4,5
160 DATA 8,2,4,5, 8,12,3,15, 0,1,4,15
170 DATA 8,12,3,5, 8,5,4,15, 8,9,4,5, 8,7,4,15
180 DATA 8,12,3,5, 8,5,4,15, 8,9,4,5, 8,7,4,5
190 DATA 8,9,4,5, 8,10,4,5, 8,12,4,5
200 DATA 8,9,4,5, 8,5,4,5, 8,7,4,15
210 DATA 8,12,3,5, 8,5,4,15, 0,0,0,0
```

Using the noise channel requires a thorough understanding of sound theory. Appendix C provides an explanation of the SOUND and WAVE commands.

STORING INFORMATION ON DISK

ST BASIC stores information in files held on your diskettes. Just as in a physical file, you have to OPEN a disk file to INPUT information from it, or PRINT information to it.

ST BASIC uses the following commands to control data access:

OPEN
INPUT
PRINT#
CLOSE
INPUT#
LINE INPUT#
WRITE#

The most common uses of these commands are explained in the section on "Sequential Files." For a detailed explanation of each command, refer to Appendix C.

Sequential Files

Sequential files are files whose records must be read in the order they were written. Their structure can be compared to a stack of file cards that has no dividers. To find a particular card, you must look through the stack in sequential order, starting from the first card.

The following program example shows how to output names to a file:

```
10 CLEARW 2
20 OPEN "O", #1, "TESTFILE" ' "O" FOR OUTPUT
30 WHILE NAME$ <> "*"
40 INPUT# 1, NAME$
50 PRINT# 1, NAME$: ' ***** PRINT# PUTS DATA INTO FILE
60 WEND
70 CLOSE #1
80 END
```

This program takes input from the keyboard and sends it to a disk file named TESTFILE.

Line 20 opens TESTFILE as file number 1. The O tells ST BASIC to open TESTFILE to prepare it for output from the computer.

PRINT# 1 sends NAME$ to the disk drive just as the LPRINT command sends it to a printer.

The WHILE loop ends when you enter an asterisk (*) in the program. Line 80 closes the file. Look at the program carefully. In the next program example, you'll learn how to receive INPUT from a disk file.

The file created by the following program is a sequential file. You must read its contents in the same order as they were put into the file.

The following program reads the files from the disk:

```
10 CLEARM 2
20 OPEN "I", #1, "TESTFILE" ' "I" FOR INPUT
30 INPUT# 1, NAME$
40 WHILE NOT EOF(1): ' ***** EOF MEANS END OF FILE
50 PRINT NAME$
60 INPUT# 1, NAME$
70 WEND
80 CLOSE #1
90 END
```

Line 20 opens TESTFILE, as file number 1, to prepare it for INPUT to the computer. "I" means input.

The input loop uses the same technique you've used with DATA statements. Instead of DATA, you use the INPUT# statement. The WHILE loop ends when file #1 from the OPEN statement meets the EOF (End of File) condition.

When the loop is finished, line 80 closes the file.

Sequential files have the following important features:

- When you OPEN a file for output, its current contents are lost.
- If you open a nonexistent file for output, OPEN will create a file using the name you give.
- If you attempt to OPEN a nonexistent file for INPUT, ST BASIC will display an Error Message.

- Files are assigned numbers to make it easier to refer to them in a program. Up to three different files can be open at once.

Random Access Files

ST BASIC can also create files that are accessed randomly by record. A random access file can be compared to a stack of index cards with labeled dividers. To find the first, tenth, or hundredth card, you go directly to the divider labeled one, ten, or one hundred.

You can search these files more rapidly than sequential files. Using random access files requires an understanding of numeric variables, arrays, and strings. These are described in the following sections. The last section, "Advanced Concepts," provides more details about random access files.

Note: When you enter records to be saved to disk as a random access file, the first record must be number 1 (one) and all subsequent records must be incremented by 1 (one). This is most easily accomplished by using a FOR . . . TO . . . NEXT loop (e.g., FOR RECORD = 1 to 2000; IF RECORD = 0 THEN CLOSE #1:END).

NUMERIC VARIABLES

Most programs work with units of data expressed as numbers or strings. Names are usually assigned to units of data to help clarify their use. For example, you might want to total a list of sales and expenses by saying that TOTAL = TOTAL + SALES − EXPENSES. Since the values of SALES, EXPENSES, and the TOTAL vary, they are variables.

You can enter a string of data, such as a name, into a string variable. And remember that a string is any kind of data, such as text, that is not to be interpreted as a number.

ST BASIC can represent numbers as real numbers and as integers.

Real numbers can have fractional values. Pi (3.1416) is a real number. So is 5.00, because it has a decimal point in its expression.

Integers are whole numbers with no fractional parts. The number 5 is an integer; so is 5,729,346.

If you don't specify a variable as either a real number or an integer, ST BASIC interprets it as an integer. ST BASIC also converts integers to real numbers whenever necessary.

You can specify numeric type using % and !.

TOTAL%	An integer variable.
TOTAL!	A real number variable.

When you declare a variable's type, ST BASIC does not convert it to another type unless you designate the change.

You can convert a numeric variable's type using the following statements:

CINT (TOTAL)	Converts TOTAL to an integer.
BALANCE = INT (TOTAL)	The integer value of TOTAL is assigned to BALANCE, but TOTAL is unchanged.
BALANCE = FIX (TOTAL)	BALANCE is assigned the whole number part of TOTAL with any decimal part removed. The decimal isn't rounded; it's cut off.

Arrays

Any arrangement of information into rows and columns is called an array.

Whenever you arrange numbers into rows and columns, as in a spreadsheet, you create an array. A printed page of text is an array of characters (rows) and lines (columns). You can arrange variables into arrays by defining their row and column limits when you name them.

```
10 CLEARW 2: COUNT=0
20 INPUT "ENTER SCORE: ", SCORE
30 WHILE SCORE > 0
40 TALLY(COUNT)=SCORE
50 INPUT "ENTER SCORE: ", SCORE
60 COUNT=COUNT+1
70 WEND
80 CLEARW 2
90 FOR I=0 TO COUNT
100 AVERAGE=AVERAGE+TALLY(I): NEXT:
AVERAGE=AVERAGE/COUNT
110 PRINT "THE AVERAGE FOR ";COUNT;" SCORES IS:
";AVERAGE
```

```
120 PRINT: INPUT "WHICH SCORE WOULD YOU LIKE TO SEE";
SELECTION
130 WHILE SELECTION > 0 AND SELECTION < COUNT+1
140 PRINT "SCORE ";SELECTION;" IS:
";TALLY(SELECTION)
150 PRINT: INPUT "WHICH SCORE WOULD YOU LIKE TO SEE";
SELECTION
160 WEND
170 END
```

Enter up to 10 scores. When you are through, enter a 0 or a negative score to exit the entry loop (see line 30).

Using an array allows you to keep many entries in memory and access them easily.

In the previous program, TALLY is an array with one dimension (i.e., one row of several columns). Arrays can have more than one dimension. If you wanted to make a record of sales and commissions, you could write a program similar to the following example:

```
10 CLEARW 2: COUNT=0
20 INPUT "ENTER SALES: ", SALES(COUNT,0):
' ***** INPUT DIRECTLY TO ARRAY *****
30 WHILE SALES(COUNT,0) > -0
40 SALES(COUNT,1)=SALES(COUNT,0)*0.08:
' ***** COMPUTE COMMISSION *****
50 INPUT "ENTER SALES: ", SALES(COUNT,0)
60 COUNT=COUNT+1
70 WEND
80 CLEARW 2
90 PRINT "ENTRY   SALES   COMMISSION"
100 FOR I=0 TO COUNT
110 PRINT I, SALES(I,0), SALES(I,1)
120 NEXT
130 END
RUN
```

DIM

ST BASIC establishes the size of an array the first time you refer to it. The default size of an array is 10 rows by 10 columns (10,10). You can declare your own array dimensions with the DIM statement. DIM TOTAL (5,50) defines TOTAL as an array with 5 rows and 50 columns. DIM RECORD (5,50,100) defines RECORD as an array of 5 rows, 50 columns, and 100 planes. Each number within the parentheses refers to a dimension. ST BASIC allows up to 15 dimensions in an array. The number of elements and dimensions you can use depends on the amount of memory you have available for a particular program.

Option Base

The arrays in the previous programs start at element (0,0). This means that item 1 is in position (0,0). It is awkward to address an element as (0,0). ST BASIC allows you to specify (1,1) as the base element of an array. Look at the following programs:

```
10 OPTION BASE 1
20 DIM POINTS(10,5)
```

The first element of the array POINTS is (1,1).

```
30 OPTION BASE 0
40 DIM VALUES(10,5)
```

The first element of the array VALUES is (0,0).

Both arrays are the same size.

Character Strings

A string is a special type of array. The first dimension of a string is its length. DIM A$(40) sets a string length of 40 characters. The following ST BASIC reserved words manipulate strings:

LEFT$
LSET
RIGHT$
RSET
MID$

Enter this program:

```
10 CLEARW 2
20 A$="JAN TOM BRANDON"
30 B$=LEFT$ (A$,3): PRINT B$: ' ***** PRINTS LEFT
3 CHARACTERS *****
40 B$=MID$ (A$,5,3): PRINT B$: ' ***** PRINTS 3
CHARACTERS STARTING AT CHARACTER 5 *****
50 B$=RIGHT$ (A$,7): PRINT B$: ' ***** PRINTS RIGHT
7 CHARACTERS *****
```

LSET and RSET left-justify and right-justify strings. Enter this example:

```
10 FULLW 2: CLEARW 2
20 A$="          ": ' 20 SPACES
30 B$="JOHN PHILLIP SOUZA"
40 LSET A$=RIGHT$ (B$,7)
50 PRINT A$
60 RSET A$=LEFT$ (B$,4)
70 PRINT A$
80 END
```

LSET and RSET change the content of A$ without changing its length. Use them to maintain fixed field lengths in data entry routines and random access disk files.

LEFT$ accesses characters from the left portion of a string. You can assign the characters to another string or print them without changing the string.

MID$ and RIGHT$ take characters from the mid and right portions of the string. They can be assigned to another string or printed.

Converting String Variables

You can convert string variables to numeric variables using STR$:

```
A$=STR$ (TOTAL)
```

The VAL function converts numeric variables back to string variables:

```
TOTAL=VAL (A$)
```

After a program finishes performing computations with numbers, you may want to include those numbers in a word processing program. In a text (word processing) file, it's easier to treat the numbers as strings. Or you may want to perform computations with numbers taken from a text file. The STR$ and VAL functions convert values so that you can use them in both text and data files.

ADVANCED CONCEPTS

ST BASIC lets you incorporate custom functions, chained overlays, and custom assembly-language modules into your programs.

DEF FN

Occasionally you may need a function that doesn't exist in ST BASIC. You can define your own functions in ST BASIC with the DEF FN statement. Enter this program:

```
10 DEF FNCOST=QUANTITY * PRICE
20 INPUT "QUANTITY: ",QUANTITY
30 INPUT "PRICE: ", PRICE
40 PRINT "COST IS "; FNCOST
50 END
```

This program defines FNCOST as the result of the computation:

QUANTITY * PRICE

You can also define a function and pass a value to it. Enter this program:

```
10 DEF FNPAY (SALES)=SALES * COMMISSION - 0.02*SALES
20 INPUT "COMMISSION: ", COMMISSION
30 INPUT "SALES: ", SALES
40 WHILE SALES > 0
50 PRINT "PAY IS "; FNPAY (SALES)
60 INPUT "SALES: ", SALES
70 WEND
80 END
```

In this program, the number you assign to FNPAY will be used to compute the function result. Notice that in both program examples, global variables like QUANTITY or COMMISSION get their values from outside the function statement. The same is true for predefined functions such as +, where the result of LENGTH + WIDTH would depend on previously set values for LENGTH and WIDTH.

Chaining Programs

One ST BASIC program can run another one. The program line:

```
100 CHAIN "NEXTPROG"
```

causes the program NEXTPROG.BAS to be loaded and run, replacing the current program and destroying all current variables.

The program line

```
100 CHAIN MERGE "NEXTPROG"
```

loads NEXTPROG.BAS, but leaves current variables intact. CHAIN MERGE replaces current program lines with similarly numbered lines in NEXTPROG.BAS. The program does not replace the lines whose numbers are not duplicated in NEXTPROG.BAS.

The following programming rules apply to CHAIN and CHAIN MERGE:

- CHAIN destroys the current program and variable values.
- CHAIN MERGE does not alter variable values. It only replaces duplicate lines.

Binary Files

A binary file is made up of data bytes that are not interpreted as either numbers or text as they are read into memory. They are loaded as 8-bit bytes into memory.

You can save parts of memory directly to disk (in binary format) or read binary files directly into memory. This is useful for saving and loading screen images and machine language modules.

A machine language module is a binary file containing program codes that can be read and executed directly by the 68000 microprocessor in your ST Computer.

The program line:

```
BSAVE "ARRAY",100,650
```

saves 650 bytes, beginning at address 100, to a file named ARRAY.

The program line:

```
BLOAD "ARRAY",100
```

loads the file ARRAY into memory beginning at address 23.

If your binary file is a machine language module, you can run it from ST BASIC using the CALL command. Look at the following example:

```
100 PLOT=23
110 CALL PLOT(X,Y,Z)
```

The lines assign the starting address of a routine to the variable PLOT and tell ST BASIC to run the routine, passing the variables X, Y, and Z to the program.

Refer to Appendix E for more information about how to use assembly language modules with ST BASIC.

Random Access Files

You can access information more quickly in a random access file than a sequential file. Random access files must have fixed-length fields because they may use more memory to store text than sequential files. However, random access files are more efficient for storing numbers than sequential files because they store numbers in binary format, whereas sequential files use ASCII format.

Note: ASCII stands for American Standard Code for Information Interchange. ASCII is the standard translation code of computer data into printable characters.

The following reserved words control random access files:

OPEN
CLOSE
EOF
LSET
RSET
FIELD
PUT
GET
LOC

OPEN, CLOSE, and EOF are familiar to you from the section on sequential files. LSET and RSET were described in the section on strings. FIELD, PUT, GET, and LOC are new commands.

FIELD specifies the format of a random access file. Each record in the file will consist of the data FIELD you specify.

You PUT records into a file and GET records from a file. Each record consists of the same number and types of fields. The fields are defined in the FIELD statement of the program that created the file. Any program reading the records must define records that match those already in the file. (The number and types of fields must be the same.) You do not have to use the same variable names in each program.

LOC lets you know your location in a file. A = LOC(1) sets A equal to the number of bytes read from a sequential file or the current record number in a random access file.

The following program shows how to access and modify records in a random access file:

```
10 ' ***** RPUTGET.BAS BY RUSS G.
20 CLEARW 2: FULLW 2
30 OPEN "R", #1, "TESTFILE": ' ***** OPEN FOR RANDOM
ACCESS
35 ' ***** EACH RECORD TO HAVE 3 FIELDS
40 FIELD #1,20 AS V$,10 AS X$,30 AS N$: ' 20 SPACES
RESERVED FOR V$,10 FOR X$, ETC.
45 ' ***** LEFT JUSTIFY V$ AND X$
50 LSET V$="HELLO"
60 LSET X$="THIS"
65 ' ***** RIGHT JUSTIFY N$
70 RSET N$="IS RECORD 1"
75 PRINT V$;X$;N$
80 PUT #1: ' ***** PUT FIRST RECORD INTO FILE
90 RSET N$="IS RECORD 2"
95 PRINT V$;X$;N$
100 PUT #1: ' ***** PUT SECOND RECORD INTO FILE
110 CLOSE 1
115 PRINT
120 OPEN "R", #1, "TESTFILE"
```

```
130 FIELD #1,20 AS A$,10 AS B$,30 AS C$: ' ***** NOTICE
NEW VARIABLE NAMES
140 GET #1,2: ' ***** GET SECOND RECORD FIRST
150 PRINT A$;B$;C$
160 GET #1,1: ' ***** GET FIRST RECORD NEXT
170 PRINT A$;B$;C$
180 CLOSE 1
190 END
```

You can retrieve records from the file in any order.

The following program adds a record to the end of the file:

```
10 ' ***** RAPPEND.BAS
20 CLEARW 2: FULLW 2
30 OPEN "R", #1, "TESTFILE": ' ***** OPEN FOR RANDOM
ACCESS
35 ' ***** EACH RECORD TO HAVE 3 FIELDS
40 FIELD #1,20 AS V$,10 AS X$,30 AS N$
50 LSET V$="HELLO"
60 LSET X$="THIS"
65 ' ***** RIGHT JUSTIFY N$
70 RSET N$="IS RECORD 3"
75 PRINT V$;X$;N$
80 PUT #1,3: ' ***** PUT RECORD INTO FILE
90 CLOSE 1
100 OPEN "R", #1, "TESTFILE"
110 FIELD #1,20 AS A$,10 AS B$,30 AS C$
120 FOR I=1 TO 3
130 GET #1,I: ' ***** GET RECORD
140 PRINT V$;X$;N$: ' ***** PRINT RECORD
150 NEXT
160 CLOSE 1
170 END
```

The next program replaces a record in the middle of the file:

```
10 ' ***** RINSERT.BAS
20 CLEARW 2: FULLW 2
30 OPEN "R", #1, "TESTFILE": ' ***** OPEN FOR RANDOM
ACCESS
35 ' ***** EACH RECORD TO HAVE 3 FIELDS
40 FIELD #1,20 AS V$,10 AS X$,30 AS N$
50 LSET V$="HELLO"
60 LSET X$="THIS"
65 ' ***** RIGHT JUSTIFY N$
70 RSET N$="IS THE MIDDLE RECORD"
75 PRINT V$;X$;N$
80 PUT #1,2: ' ***** REPLACE SECOND RECORD
90 CLOSE 1
100 OPEN "R", #1, "TESTFILE"
110 FIELD #1,20 AS A$,10 AS B$,30 AS C$
120 FOR I=1 TO 3
130 GET #1,I: ' ***** GET RECORD
140 PRINT V$;X$;N$: ' ***** PRINT RECORD
150 NEXT
160 CLOSE 1
170 END
```

Numbers In Random Access Files

In a random access file, it's most efficient to represent numbers in pure binary form. The following ST BASIC reserved words convert integers and real numbers to and from binary format:

MKI$	Converts integers to 2-byte strings.
MKS$	Converts real numbers to 4-byte strings.
MKD$	Converts real numbers to 8-byte strings.
CVI	Converts 2-byte strings to integers.
CVS	Converts 4-byte strings to real numbers.
CVD	Converts 8-byte strings to real numbers.

The following program shows why binary numbers can be used more efficiently than ASCII-formatted numbers in a random access file:

```
10 ' RPUTNUM.BAS
20 FULLW 2: CLEARW 2
30 OPEN "R", #1, "TESTFILE"
40 FIELD #1,20 AS NAME$,2 AS AGE$,30 AS JOB$
50 INPUT "NAME: ", N$
60 WHILE N$ <> "*"
70 INPUT "AGE: ", AGE
80 INPUT "JOB: ", J$
90 LSET NAME$=N$: LSET JOB$=J$
95 AGE$=MKI$(AGE): ' ***** CONVERT AGE TO 2 BYTE STRING
100 PUT #1
110 INPUT "NAME: ", NAME$
120 WEND
130 CLOSE 1
140 PRINT
150 OPEN "R", #1, "TESTFILE"
160 FIELD #1,20 AS NAME$,2 AS AGE$,30 AS JOB$
170 FOR I=1 TO 3
180 GET #1,I
190 AGE=CVI(AGE$): ' ***** CONVERT AGE$ TO INTEGER AGE
200 PRINT NAME$;AGE;JOB$
210 NEXT
220 CLOSE 1
230 END
```

Note: Random access files can have no more than 4096 bytes in a record and 32767 records in a single file.

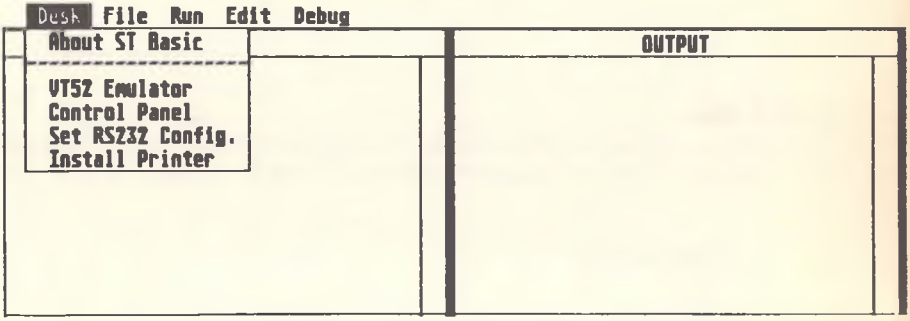
You should now have a good understanding of the vocabulary and syntax of ST BASIC. Remember to use the Appendices to learn more about the language. Using Appendices A, B, and C will provide you with all the information you need to experiment with ST BASIC and your ST Computer.

CHAPTER 3
ST BASIC MENUS

This chapter describes the menus available from the ST BASIC Desktop. The Menu headings are Desk, File, Run, Edit and Debug. They are located along the top edge of the ST BASIC Desktop in the Menu Bar. Each heading has its own menu. To access a menu, point at the menu heading. The word will become shaded and the menu will pop down. If you don't want to select a menu item, click anywhere else on the ST BASIC Desktop and the menu will pop back into the Menu Bar.

DESK

The Desk menu contains options that are available from ST BASIC and from within most applications programs that run on the ST Computer.



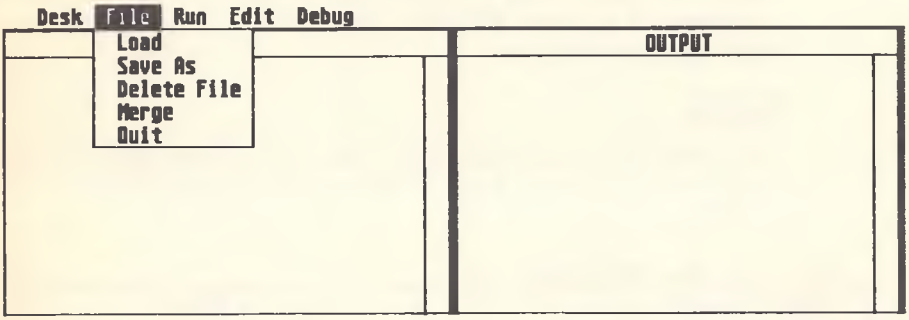
About ST BASIC

This option is the billboard for the applications program. Copyright and general program information are displayed. Select the About ST BASIC option and the following Dialog Box is displayed:

The other options in the Desk menu—VT™52 Emulator, Control Panel, Set RS232 Config., and Install Printer—are explained in detail in the *ATARI ST Owner's Manual*. Refer to the section on each option in Chapter 5 of that manual.

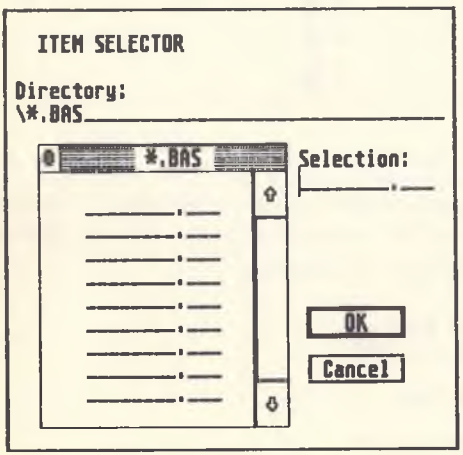
FILE

The File menu contains options that let you read information from or write information to the disk drive.



Load

The Load option reads a file that has been stored on a floppy disk. Select the Load option and the following Dialog Box is displayed:



To select a file listed in the Item Selector Box, point at a filename and double-click the left mouse button. You can also select a file by clicking once on an item and then clicking once on the Ok button.

The current directory is displayed at the top of the Item Selector Box under the heading "Directory". If the file you want to access is stored in a different directory, you can change the directory. Click on the Directory heading, use the **[Backspace]** key to erase the current directory name, and type in the name of the directory you want to use. To view a listing of the files under the new directory, simply click anywhere inside the Directory Window and the new directory listing will appear.

Note: One hundred out of a possible 112 files can be displayed at one time in the Directory Window. Use the Scroll Bar to view the remaining directory listings.

If you decide not to load a particular file, or if the file you want is not present, you can exit the Item Selector Box by clicking on the Cancel button.

Save As

The Save As option creates a new file. You can also use this option to make a copy of a file using a different name. Each time you select the Save As option, the Item Selector Box will be displayed.

To enter the filename of a new file in the Item Selector Box, type the filename on the line indicated in the Item Selector Box. If you are replacing an existing file, select the name from the list of filenames in the directory.

Note: The Save As option executes a REPLACE command, replacing the program on disk with the program of the same name in memory.

Delete File

The Delete File option deletes an ST BASIC program from a floppy disk. When you select the Delete File option, the Item Selector Box will be displayed. To delete a file, either double-click on the filename, or click on the filename once and then click on the Ok button.

Merge

The Merge option loads an ST BASIC program into memory without erasing the program already in memory. It is most often used to merge small program modules into one larger program.

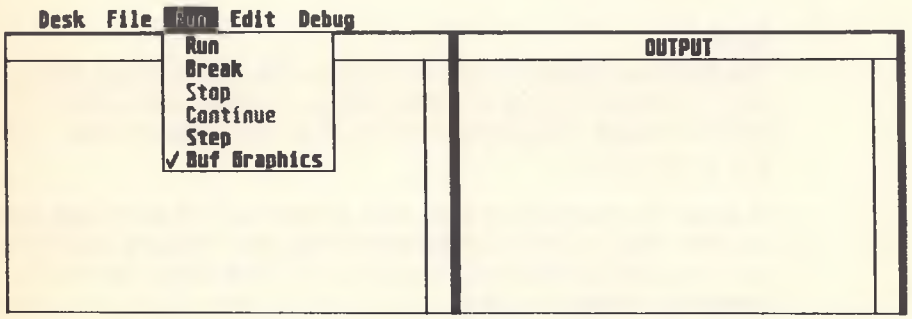
When you select the Merge option, the Item Selector Box will be displayed. Select the program to be merged and click on the Ok button. List the program to insure that the program segments have been merged properly.

Quit

The Quit option lets you exit ST BASIC and returns you to the GEM Desktop. When you use this option, the program in memory is not saved before exiting ST BASIC.

RUN

The Run menu provides options that are used to control the starting and stopping of the procedures you use with ST BASIC.



Run

The Run option runs the ST BASIC program currently in memory.

Break

The Break option halts an ST BASIC program. If you are in Edit mode, selecting the Break option from the Run menu returns control of the program to the screen editor. Selecting the Break option in any other circumstance places the program in Break mode. While in Break mode, you can pause, examine and modify variables, or continue program operation.

Stop

The Stop option halts program execution from within the Break mode.

Continue

The Continue option restarts program execution from the position where the Break option was selected.

Step

The Step option lets you execute a program procedure one step at a time. When you select the Step option, the following Dialog Box is displayed:

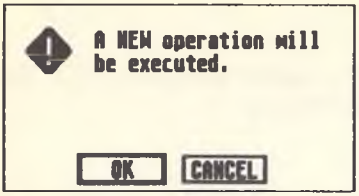
Click on the Ok button to acknowledge the Step procedure. Each time you press the [Return] key, the next program line is executed. Press [Control] [G] to exit the Step option.

Buf Graphics

When the Buf Graphics option is implemented, memory space is reserved for graphics created in the Output Window. A buffer space is established. The buffer takes up 32,000 bytes of memory and reduces the amount of memory space available for program data.

The Buf Graphics option is implemented when the check mark is in front of the option in the Run menu. If you want to select or turn off the Buf Graphics option, click on the Buf Graphics option.

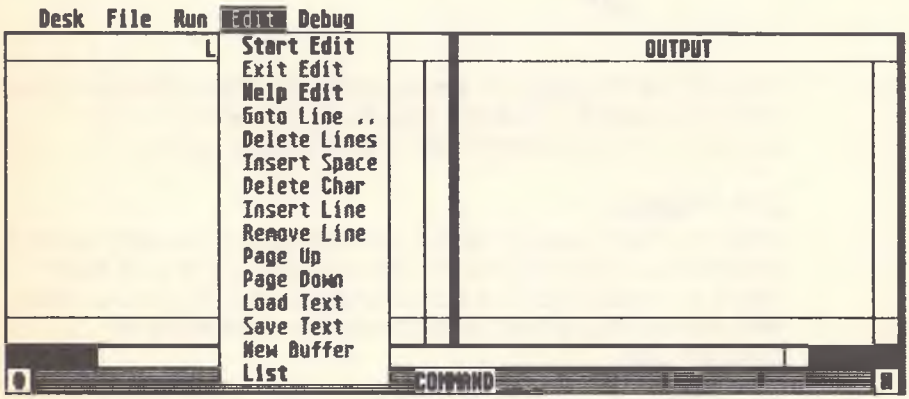
When you select or turn off the Buf Graphics option, the following Dialog Box is displayed:



Click on one of the exit buttons to either select or turn off the buffer. ST BASIC will execute a NEW command and erase the program memory each time the buffer is turned on or off.

EDIT

The Edit menu controls the editing capabilities of ST BASIC.



Start Edit

The Start Edit option lets you enter the Edit mode.

Exit Edit

The Exit Edit option lets you exit the Edit mode.

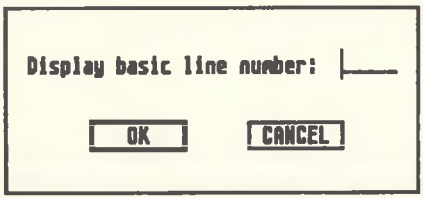
Help Edit

Select the Help Edit option and the following Dialog Box is displayed:

Each item in the Help Edit Dialog Box corresponds to an option in the Edit menu. The function key designations—F1 through F10—correspond to the function keys on the ST Computer keyboard. Each function can be accomplished either through the menu option or by pressing the correct function key.

Goto Line

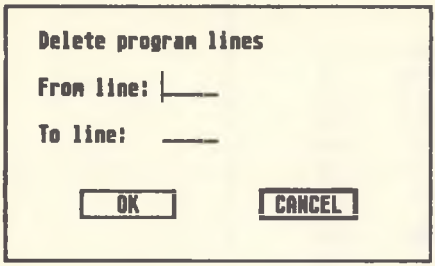
The Goto Line option moves the cursor to the line number you enter in the Dialog Box. Select the Goto Line option and the following Dialog Box is displayed:



Type in the line number you want the cursor moved to. Click on either the OK or CANCEL button to exit to the Edit mode.

Delete Lines

The Delete Lines option deletes the range of program lines you enter in the Dialog Box. Select the Delete Lines option and the following Dialog Box is displayed:



Type in the beginning program line number you want deleted, then point to the heading "To line" and type in the ending line number. Click on either the OK or CANCEL button to exit to the Edit mode.

Insert Space

The Insert Space option inserts a space at the cursor position.

Delete Char

The Delete Char option deletes the character underneath the cursor and moves all characters to the right of the cursor one space to the left.

Insert Line

The Insert Line option inserts a blank line at the cursor position.

Remove Line

The Remove Line option removes the line at the cursor position. After the Remove Line option is selected, the program statement is shown in lighter text. Placing the cursor on this line and pressing [Return] adds the statement to the program. If you want to remove the line from the program and the screen, place the cursor on the line with lighter text and click on the Remove Line option.

Page Up

The Page Up option moves the cursor to the top of the previous window in the program.

Page Down

The Page Down option moves the cursor to the top of the next window in the program.

Load Text

The Load Text option loads the contents of a file named BASIC.BUF into the Edit Window.

Save Text

The Save Text option places the text from the current Edit Window into a file named BASIC.BUF. Twenty-four lines of text are saved with this option. The text, however, does not have to be an ST BASIC program.

New Buffer

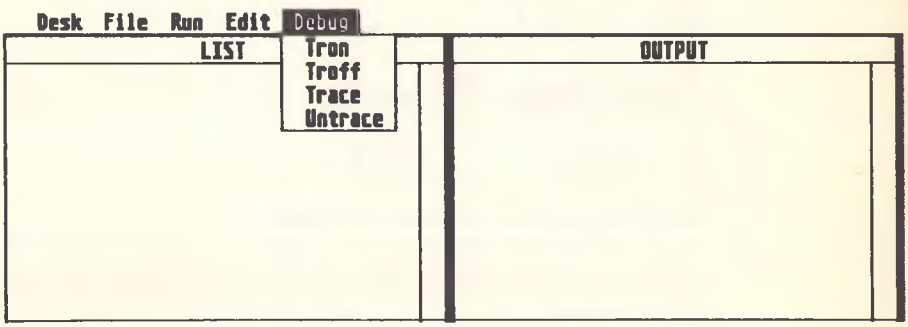
The New Buffer option places the current ST BASIC program into the Edit Window.

List

The List option places the current ST BASIC program into the List Window.

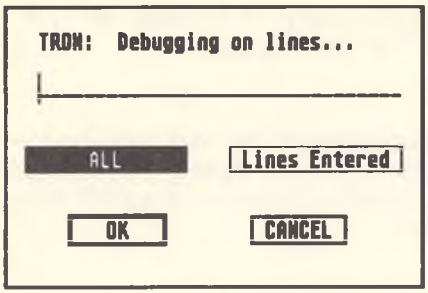
DEBUG

The Debug menu controls the debugging features of ST BASIC that are controllable through the ST BASIC menus.



Tron

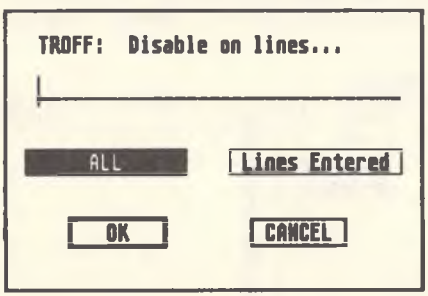
The Tron option of ST BASIC prints the current line number as the ST BASIC program is running. Select the Tron option and the following Dialog Box is displayed:



Type in the line numbers of the program lines that you want to follow as the program is running. Click on the Ok button in the Dialog Box.

Troff

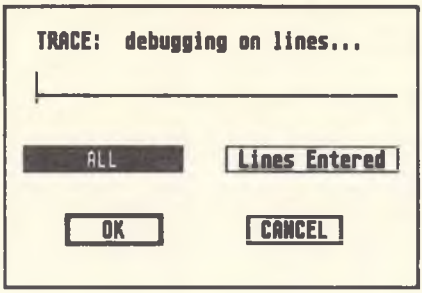
The Troff option turns off the Tron option. Select the Troff option and the following Dialog is displayed:



Click on the Ok button in the Dialog Box and the Tron option will be turned off.

Trace

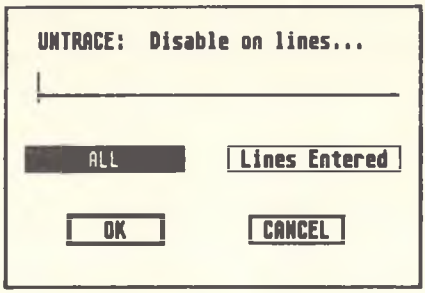
The Trace option prints the current program line as the ST BASIC program is executed. Select the Trace option and the following Dialog Box is displayed:



Type in the line numbers you want to trace in the Dialog Box. Click on the Ok button to start the Trace option.

Untrace

The Untrace option turns off the Trace option. Select the Untrace option and the following Dialog Box is displayed:



Click on the Ok button in the Dialog Box and the Trace option will be turned off.

APPENDIX A
ST BASIC RESERVED WORDS

The reserved words used in ST BASIC are listed below. If you use any of these words as a variable name, the Error Message, "Something is wrong," will appear on the screen.

ABS
ALL
AND
AS
ASC
ATN
AUTO
BASE
BLOAD
BREAK
BSAVE
CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLEARW
CLOSE
CLOSEW
COLOR
COMMON
CONT
CONTRL
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DEF FN
DEFDBL
DEFINT
DEFSEG
DEFSNG
DEFSTR
DELETE
DIM
DIR
DO
EDIT
ELLIPSE
ELSE
END
EOF
ERA
ERASE
ERL
ERR
ERROR
EQV
EXP
FIELD
FIELD#
FILL
FIX
FLOAT
FOLLOW
FOR
FRE
FULLW
GB
GEMSYS
GET
GET#
GO
GOSUB
GOTO
GOTOXY
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
INTIN
INTOUT
KILL
LEFT$
LEN
LET
LINE
LINEF
LIST
LLIST
LOAD
LOC
LOF
LOG
LOG10
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT

NOT
OCT$
OLD
ON
OPEN
OPENW
OPTION
OR
OUT
PCIRCLE
PEEK
PELLIPSE
POKE
POS
PRINT
PRINT#
PTSIN
PTSOUT
PUT
QUIT
RANDOMIZE
READ
REM
RENUM
REPLACE
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SEG
SGN
SIN
SOUND
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSDBG
SYSTAB
SYSTEM
TAB
TAN
THEN
TO
TRACE
TROFF
TRON
UNBREAK
UNFOLLOW
UNTRACE
USING
VAL
VARPTR
VDISYS
WAIT
WAVE
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

APPENDIX B
OPERATORS, ORDER OF PRECEDENCE, AND FUNCTION SUMMARY

LOGICAL OPERATORS

The logical operators recognized by ST BASIC are NOT, AND, OR, XOR, IMP, and EQV. These logical operators work on the flags resulting from logical expressions. A TRUE flag equals -1 and a FALSE flag equals 0. Thus the statement **A = 1: B = 2: PRINT A = B** prints 0, while the statement **A = 1: B = 2: PRINT A < >B** prints -1.

The result of AND is TRUE when both arguments are TRUE: **2 + 2 = 4 AND 3 + 2 = 5** is TRUE.

The result of OR is TRUE when either argument is TRUE: **2 + 2 = 4 OR 3 + 2 = 7** is TRUE.

IMP is the abbreviation for implication. IMP works on logical expressions to check the validity of premises and conclusions. IMP is TRUE in all cases except where a premise is TRUE and a conclusion is FALSE.

The statement **2 + 2 = 4 IMP 3 + 2 = 6** is FALSE.

The following statements are valid implications and are considered TRUE:

```
2 + 2 = 4 IMP 3 + 3 = 6
2 + 2 = 3 IMP 3 + 3 = 6
2 + 2 = 3 IMP 3 + 3 = 7
```

The following operators work bitwise on single byte integer numbers according to the following:

AND produces a result in which a bit is equal to 1 only where there is a 1 in both arguments. **A% = 5: B% = 3: C% = A% AND B%** makes C% equal 1.

OR produces a result in which a bit is equal to 1 where there is a 1 in either argument. **A% = 5: B% = 3: C% = A% OR B%** makes C% equal 7.

XOR produces a result in which a bit is equal to 1 where there is a 1 in either argument, but not in both arguments. **A% = 5: B% = 3: C% = A% XOR B%** makes C% equal 6.

EQV produces a result where a bit is equal to 1 where there is a 1 in both arguments, or where there is a 0 in both arguments. A bit is equal to 0 where the bits in the argument differ. **A% = 5: B% = 3: C% = A% EQV B%** makes C% equal -7.

Truth Table

NOT

X	NOT X
0	1
1	0

AND

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

IMP

X	Y	X IMP Y
0	0	1
0	1	1
1	0	0
1	1	1

EQV

X	Y	X EQV Y
0	0	1
0	1	0
1	0	0
1	1	1

ARITHMETIC OPERATORS

Symbol	Name	Example
+	Addition	X + Y
–	Subtraction	X – Y
*	Multiplication	X * Y
/	Division	X / Y
MOD	Modulus	X MOD Y
∧	Exponentiation	X ∧ Y

RELATIONAL OPERATORS

Symbol	Meaning	Example
=	Equals	X = Y
< >	Does not equal	X < > Y
<	Is less than	X < Y
>	Is greater than	X > Y
< =	Is less than or equal to	X < = Y
> =	Is greater than or equal to	X > = Y

ORDER OF PRECEDENCE FOR OPERATORS

Operator	Explanation
( )	Items in parentheses have highest priority
∧	Exponentiation
–	Negation
*	Multiplication
/	Floating-point and integer division
MOD	Modulus
+, –	Addition, subtraction
=, < >	
<, >	Relational operators
< =, > =	
NOT, AND	
OR, XOR	Logical operators, in order given
IMP, EQV	

SUMMARY OF ST BASIC FUNCTIONS

Functions operate on constants and variables to produce values for variables. A constant is a number, such as 250.4 or a string such as HELLO. A variable is a named numeric value, such as TOTAL or a named string value, such as NAME$.

Variable Names

Variable names cannot contain spaces. They can be as long as you like, but only the first 31 characters are used by ST BASIC to distingush them from one another.

Numeric Variables

There are different types of numeric variables. The following table summarizes variable types:

Character	Type	Example
$	String	NAMES$
%	Integer	RECORD.NUMBER%
!	Real Number	TOTAL.PROFIT!

Type Declarations

The following statements declare variable types in ST BASIC (see definitions in Appendix C):

DEFSTR declares string variables.
DEFINT declares integer variables.
DEFSNG declares real number variables.

Numeric Functions

The numeric functions available in ST BASIC are shown below:

Function	Explanation
ABS	Returns the absolute value of a number.
ATN	Returns the arctangent of a number.
COS	Returns the cosine of a number.
EXP	Returns to the power of a given value.
LOG	Returns the natural logarithm of a number.
LOG10	Returns the base-10 logarithm of a number.
RND	Generates a sequence of random numbers.
SIN	Returns the sine of a number in radians.
SQR	Returns the square root of a number.
TAN	Returns the tangent of a number in radians.

String Functions

Strings may be concatenated using + as in A$ = B$ + C$. Other string functions are available in ST BASIC as shown in the following table.

Function	Explanation
INSTR	Finds the first occurrence of a particular sequence of characters within a string and returns its position.
LEFT$	Returns the leftmost characters in a string.
LEN	Returns the number of characters in a string.
MID$	Extracts a string from within a string, beginning at whatever point you specify.
RIGHT$	Returns the rightmost characters in a string.
SPACE$	Returns a string of spaces.
STR$	Converts a number to a string.
STRING$	Returns a string of a given length.

Arrays

ST BASIC supports numeric and string arrays. The DIM statement dimensions the variables. When referencing arrays, subscripts refer to rows, columns, and planes—in that order. Subscript values may be any valid numeric constant, variable, or expression. Integer values are the most efficient, as real numbers are converted to integers when used as subscripts in an array. Arrays accept input directly and may be used as would any variable in a BASIC statement.

Two-Dimensional Array

	(0)	(1)	(2)	
(0)	(0.0)	(0.1)	(0.2)	SUN
(1)	(1.0)	(1.1)	(1.2)	MON
(2)	(2.0)	(2.1)	(2.2)	TUE
(3)	(3.0)	(3.1)	(3.2)	WED
(4)	(4.0)	(4.1)	(4.2)	THU
(5)	(5.0)	(5.1)	(5.2)	FRI
(6)	(6.0)	(6.1)	(6.2)	SAT
	6 AM	2 PM	10 PM	

The maximum number of elements in an array is limited by available memory. Elements of different data types use memory differently, as shown below:

INTEGER elements use 2 bytes.
REAL NUMBER elements use 4 bytes.
STRING elements use 6 bytes.

Line Format

The line format for ST BASIC is as follows:

<line number> <label:> <statement> <:statement> <:'remark>

The optional label may be used instead of the line numbers as the line descriptor in a GOTO or GOSUB statement.

Filename Conventions

ST BASIC program lines use the extension .BAS to identify them as BASIC programs. Filenames cannot exceed 8 characters in length and they may use an extension of no more than 3 characters. For example, the filename FILENAME.DAT is a valid filename.

APPENDIX C
COMMANDS, FUNCTIONS, AND STATEMENTS

This Appendix describes the ST BASIC commands, functions, and statements in alphabetical order. The following format is used to present each term:

- Definition of term.
- Syntax example.
- Explanation of how the term is used in ST BASIC.
- Cross reference to other ST BASIC terms.
- Program example.

The syntax formats in this section conform to the following typographical conventions:

- Words in angle brackets, < >, describe the kind of data you must insert in their places. They are self-explanatory. For example, <*variable*>, means that when you are writing a statement, you write a variable in <*variable*>'s place.

- Items enclosed in square brackets, [ ], are optional and cannot be repeated.

- Items enclosed in parentheses, ( ), are optional and can be repeated.

- Words in uppercase are ST BASIC keywords.

ABS

The ABS function returns the absolute value of a number which is always positive or zero.

Syntax:

X = ABS(<*numeric expression*>)

```
X=ABS(N)
```

Explanation:

ABS returns an integer value for an integer argument. For real numbers, the value returned has the same precision as the argument.

Example:

```
Ok 10 I%=ABS(-9)
Ok 20 PRINT I%
Ok 30 X!=ABS(325556.244)
Ok 40 PRINT X!
Ok 50 END
OK RUN
 9
 325556
Ok
```

ASC

The ASC function returns the ASCII value of the first character in a string.

Syntax:

I% = ASC(<*string expression*>)

```
I%=ASC(A$)
```

Explanation:

ASC returns an integer between 0 and 255. The string must contain at least one character. If the string expression is a null string, an error number 5 occurs.

The CHR$ function is the inverse of ASC. See Appendix E for a list of ASCII characters and corresponding numeric values.

Example:

```
Ok 10 A$="Murphy, James"
Ok 20 PRINT ASC(A$)
Ok RUN
 77
Ok
```

ATN

The ATN function returns the arctangent of a number.

Syntax:

X! = ATN(<*numeric expression*>)

```
!=ATN(N%)
```

Explanation:

The ATN function returns a real number. The number is an angle in radians that ranges from -PI/2 to PI/2. The TAN function is the inverse of ATN.

Example:

```
Ok 10 RADIANS!=ATN(0.99999)
Ok 20 PRINT "THE ANGLE IN RADIANS IS ";RADIANS!
Ok 30 PRINT
Ok 40 PI=3.14159
Ok 50 DEGREES=RADIANS! * 180/PI
Ok 60 PRINT "THE ANGLE IN DEGREES IS";CINT(DEGREES)
Ok RUN
THE ANGLE IN RADIANS IS .785393

THE ANGLE IN DEGREES IS 45
Ok
```

AUTO

The AUTO command generates a line number each time you press the [Return] key. A [Control] [G] turns AUTO off. A line number may not have a value greater than 65535. The AUTO command may not be executed from the editor.

Syntax:

AUTO [<*starting line number*>] [,<*increment*>]

```
AUTO
AUTO 50,25
AUTO ,20
AUTO 50
```

Explanation:

You specify the first line number to generate and the number to add to generate each following line number. If you do not specify the starting line number, AUTO starts at line 10. If you do not specify an increment, AUTO uses either 10 or the last increment specified by an AUTO command.

If a line number already exists, AUTO prints two asterisks before it (**10). If you enter a new program line, it will replace the original one when you press [Return]. If you simply press [Return], the old program line will remain undisturbed.

A [Control] [G] stops AUTO. But it does not perform the same function as [Return]. A [Control] [G] does not enter a program line and it will not change an existing line.

Example:

```
080k AUTO
10
20
30
 .
 .
 .

OK AUTO 50, 25
50
75
100
 .
 .
 .
```

```
Ok AUTO , 20
10
30
50
  .
  .
  .

Ok AUTO 50
50
70
90
  .
  .
  .
```

BLOAD

The BLOAD statement loads a file into memory.

Syntax:

BLOAD <*filespec*>[,<*address*>]

```
BLOAD TESTFILE.DAT, 250
```

Explanation:

BLOAD is used to load machine language programs, and arrays and their contents. BLOAD can also display screen images.

BLOAD loads a file into memory at the address you give. The filespec is the full name of the file including file type. The address is the numeric expression where you want loading to begin.

If you omit the address, the address specified with BSAVE is assumed. The file loads into the same address it came from.

BLOAD does not check addresses. Although it is possible to BLOAD anywhere, do not BLOAD over ST BASIC's data areas or your program. If you do, you will most likely crash your program.

Note: BLOAD works in conjunction with the BSAVE command.

Example:

```
Ok 110 BLOAD "ARRAY",23
```

BREAK

The BREAK command stops program execution.

Syntax:

BREAK [<*list of line numbers*>]

```
BREAK -40
BREAK 10-40
BREAK 40, 125
BREAK
BREAK 40
```

Explanation:

BREAK, by itself, causes the program to stop execution after every line. Both the program line and any output are printed. A **[Return]** or the CONT command will cause the next line to execute. This is the same as the STEP command.

If you specify line numbers, program execution stops only at the specified lines.

The UNBREAK command stops BREAK.

To exit BREAK mode, type STOP or END.

Example:

```
Ok 10 N=5
Ok 20 FOR X=1 TO 5
Ok 30 N=N-1
Ok 40 PRINT N
Ok 50 NEXT X
Ok BREAK 50
Ok RUN
 4
b  50 NEXT X
Br
```

BSAVE

The BSAVE statement saves part of memory to a file.

Syntax:

BSAVE <*filespec*>,<*address*>,<*length*>

```
BSAVE TESTFILE.DAT, 250,500
```

Explanation:

BSAVE saves machine-language programs, data, or screen images. The filespec is the name of your file and the address is a numeric expresson.

Example:

```
Ok 118 BSAVE "ARRAY" ,23,650
```

CALL

The CALL statement transfers control to a machine language subroutine.

Syntax:

CALL <*numeric variable*> [(<*parameter list*>)]

```
CALL DRAW(X, Y, Z)
```

Explanation:

The numeric variable is the starting memory address of the machine language routine. The routine can be loaded into memory using BLOAD.

The optional parameter list consists of expressions that serve as arguments to pass data between the main program and the assembly routine. The parameter list is enclosed in parentheses and must be separated by commas.

Example:

```
Ok 500 BLOAD "ASHLER",185000
Ok 550 CHART = 185666
Ok 600 CALL CHART(I%, A$, X)
```

Note: The assembler routine called using the CALL command will find two parameters on the user stack (A7). The first parameter is a 2-byte integer that specifies the number of formal parameters passed from the user's program. (In the case of the above example line 600, it will be three). The second parameter on the stack is a 4-byte pointer to an array that contains the current value of the formal parameters. Each such value occupies 4 bytes in the array regardless of the type of the formal parameter (i.e., integer, double). In each case a string variable is used as formal parameter, the 4-byte value in the array will contain a pointer to the memory location containing that string.

CHAIN

The CHAIN statement transfers control and passes variables to another program. A .BAS extender is assumed unless otherwise specified.

Syntax:

CHAIN <*filespec*>[,<*line descriptor*>][,ALL]
CHAIN MERGE <*filespec*>[,<*line descriptor*>]
[,DELETE<*line descriptor list*>]

```
CHAIN NEWPROG, 100, ALL
CHAIN MERGE NEWPROG, 100, DELETE 500-600
```

Explanation:

The program you specify in the CHAIN statement replaces the original program in memory. The program chained to is sometimes called an overlay, because it overwrites all or part of the original program. The filespec is the name of the new program. It can be any string expression of a legal filename.

The MERGE option merges a program with an existing program instead of replacing it. CHAIN MERGE saves all variables, type declarations, statements, and options. if you omit the MERGE option, you must restate all DEF statements in each newly chained program. The MERGE option overlays the statements in the new program with the statements in the original program. If some of the same line numbers in the new program are the same as in the original, the new program lines replace the original ones.

You can specify a line descriptor after the filespec indicating where to begin execution in the new program. Otherwise, execution begins with the first executable statement.

The ALL option indicates that all variables in the original program are passed to the new program. ALL is not valid with CHAIN MERGE.

If you omit the ALL option, you must use the COMMON statement to declare which variables the original program and the new program can share.

See: COMMON

Use the DELETE option only with CHAIN MERGE. The DELETE option allows you to remove parts of the old program from memory to make room for the new program. The DELETE option deletes

lines from the current program before merging the program specified by <*filespec*>. Specify the line numbers to delete after the DELETE keyword.

Example:

The following statement chains to a program named CALCS.BAS:

```
Ok 400 CHAIN "CALCS"
```

The following statement chains to the CALCS.BAS program and begins execution at line 1200. All program variables can pass from the original program to the new program.

```
Ok 400 CHAIN "CALCS", 1200, ALL
```

The following statement merges the lines from an overlay named TOTAL.OVR with the program already in memory. Execution begins at line 900. Before loading the merged file, the statement deletes the list ranging from line 900 through line 2000.

```
Ok 710 CHAIN MERGE "TOTAL.OVR", 900, DELETE 900-2000
```

CHR$

The CHR$ function returns the ASCII character that corresponds to the specified ASCII decimal value.

Syntax:

A$ = CHR$(<*numeric expression*>)

```
A$=CHR$(97)
```

Explanation:

CHR$ returns a one-character string.

The numeric expression must evaluate to a legal integer.

The ASCII value of the character returned is <*expression*> MOD 256. This means that the expression will be converted to a number between 0 and 256. If the expression is greater than 256, it will be treated as the remainder of a division by 256.

CHR$ converts real numbers to integers.

Use the CHR$ function to send special characters, such as line feeds or carriage returns, to an output device. The CHR$ function is the inverse function of ASC.

Example:

```
Ok 10 PRINT CHR$(83)
Ok 20 PRINT CHR$(100)
Ok 30 PRINT CHR$(356)
Ok RUN
S
d
d
Ok
```

CINT

The CINT function rounds a number to the nearest integer.

Syntax:

I% = CINT(<*numeric expression*>)

```
I%=CINT(N)
```

Explanation:

The numeric expression must be between -32768 to 32767. Otherwise, an overflow error occurs.

See: FIX, INT

Example:

```
Ok 10 PRINT CINT(5.2)
Ok 20 PRINT CINT(62.89)
Ok 30 PRINT CINT(-456.61)
Ok RUN
 5  63
-457
Ok
```

CIRCLE

The CIRCLE statement draws circles and arcs.

Syntax:

CIRCLE <*horizontal center,vertical center,radius*>
[<,*start angle,end angle*>]

```
CIRCLE 50,80,50
CIRCLE 50,80,50,900,1800
```

Explanation:

CIRCLE draws a circle whose center is located at the point specified by the first two parameters: horizontal center and vertical center. The positions are in pixels starting from the upper left corner of the Output Window.

The third parameter, radius, is also expressed in pixels. The horizontal and vertical pixel count is dependent upon the resolution selected and the size of the output window. The circle is drawn in the plot color (parameter 3 of the COLOR statement.)

The last two parameters, start angle and end angle, are optional. If they are not specified, CIRCLE draws a circle. If they are specified, CIRCLE draws the part of a circle that lies between them. CIRCLE draws an arc, not a solid colored pie-shaped segment. Angles are expressed in degrees times 10. You would specify 45 degrees as 450, 180 degrees as 1800, etc. Zero degrees is to the right of the window, 90 degrees is toward the top, 180 degrees to the left, and 270 degrees at the bottom. CIRCLE 100,30,30,0,3600 draws a full black circle.

See: PCIRCLE, ELLIPSE, PELLIPSE

Example:

```
Ok 10 COLOR 1,0,1: CLEARW 2
Ok 20 CIRCLE 100,50,40
Ok 30 COLOR 1,0,2
Ok 40 CIRCLE 100,50,40,300,900
Ok RUN
```

[Output Window will show black circle with 60 degree red arc at 30 degrees]

```
Ok
```

CLEAR

The CLEAR statement frees all memory used for program data without erasing the program currently in memory.

Syntax:

CLEAR

```
CLEAR
```

Explanation:

CLEAR sets all numeric variables to zero and string variables to null. The CLEAR command undefines all arrays.

Example:

The following example clears all data from memory without erasing the original program:

```
Ok CLEAR
```

CLEARW

The CLEARW statement clears ST BASIC windows.

Syntax:

CLEARW <*numeric expression*>

```
CLEARW 2
```

Explanation:

CLEARW clears the specified window. The windows are as follows:

0 = The Edit Window.
1 = The List Window.
2 = The Output Window.
3 = The Command Window.

Example:

```
Ok 10 CLEARW 2
OK 20 PRINT "HELLO"
OK RUN
```

CLOSE

The CLOSE statement closes open disk files, concluding any input or output.

Syntax:

CLOSE [#]<*file number*>

```
CLOSE
CLOSE #1
CLOSE 1,3,4
```

Explanation:

The CLOSE statement closes open files, releases the file numbers, and frees all buffer space that the files use. The files must have been opened with the OPEN statement.

The file number is the identification number you assign to a file in the OPEN statement. You can specify any number of file numbers in the optional CLOSE statement and separate file numbers with commas.

A pound sign, #, in front of the file number is optional.

File numbers can be any numeric expression. The expression must evaluate to a number between 1 and 15, the maximum number of files allowed, or a "Bad File Number" error occurs. If file numbers evaluate to real values, CLOSE converts them to integers.

If you do not specify file numbers after the keyword CLOSE, the statement closes all files that have been opened.

Note: NEW, END, RUN, LOAD, OLD, QUIT, and SYSTEM close all open files automatically. The STOP statement does not close disk files.

Example:

The following statement closes all open disk files:

```
Ok 310 CLOSE
```

The following statement closes the open disk files that have been assigned the file numbers 3 and 7:

```
Ok 600 CLOSE #3, #7
```

CLOSEW

The CLOSEW statement closes one ST BASIC window.

Syntax:

CLOSEW <*window number*>

```
CLOSEW 1
```

Explanation:

Used to close one of four ST BASIC windows. This call has to be made separately to close each window. <*Window number*> specifies windows as follows:

0 - The Edit Window.
1 - The List Window.
2 - The Output Window.
3 - The Command Window.

Note: CLOSEW does certain bookkeeping chores internal to the ST BASIC interpreter that allow the system to keep track of the window status. Therefore, do not close ST BASIC windows using direct calls to AES.

COLOR

The COLOR statement sets text, fill, and plot colors and fill patterns.

Syntax:

COLOR [<*text color, fill color, line color, index, style*>]

```
COLOR 1,0,1,1,1
```

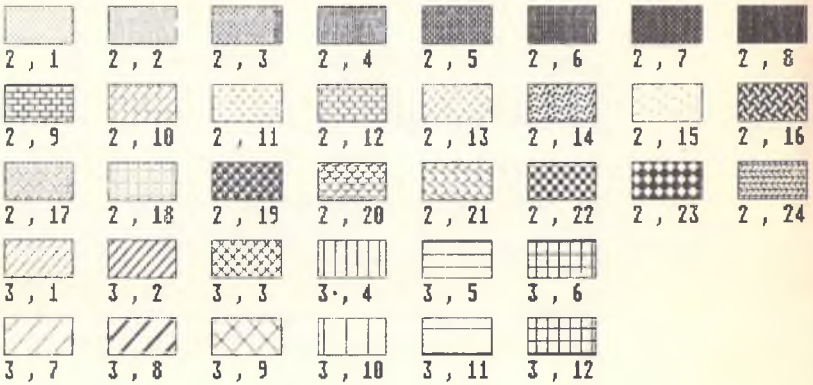
Explanation:

COLOR sets the colors of text printed to the output window, the output window background color (fill color), and the color of lines drawn in the output window as well as the color and pattern used to fill shapes. COLOR affects subsequent PRINT and graphics colors but does not change the color of text or graphics already in the output window.

The table below shows the numbers for colors in different resolutions:

Color	NUMBER	LOW	MED	HI
WHITE	0	X	X	X
BLACK	1	X	X	X
RED	2	X	X	
GREEN	3	X	X	
BLUE	4	X		
DARK BLUE	5	X		
BROWN	6	X		
DARK GREEN	7	X		
GREY	8	X		
DARK GREY	9	X		
LIGHT BLUE	10	X		
BLUE GREEN	11	X		
LIGHT PURPLE	12	X		
DARK PURPLE	13	X		
LIGHT YELLOW	14	X		
DARK YELLOW	15	X		

The following chart shows the patterns selected by parameter numbers 4 and 5 and shows the available fill styles. Under each rectangle are two numbers, separated by a comma. The number to the right of the comma corresponds to the style: Hollow, Pattern, or Hatch. The number to the left of the comma corresponds to the index for the particular pattern or hatch.



Example 1:

```
OK 10 COLOR 1,0,1
Ok 20 PRINT "BLACK"
Ok 30 COLOR 2,0,1
Ok 40 PRINT "RED"
Ok 50 COLOR 1,0,1
Ok RUN
BLACK **** IN BLACK *****
RED **** IN RED ****
Ok
```

Example 2:

```
10 COLOR 1,2,3,1,1
20 FULLW 2: CLEARW 2
30 K=(K+10) MOD 3600
40 FOR I=3 TO 11
50 COLOR 1,1,1,I,2
60 J=I*400
70 PCIRCLE 150,80,80,(J+K+3600) MOD 3600,
(J+K+400) MOD 3600
80 NEXT
90 GOTO 30
```

COMMON

The COMMON statement declares the variables that a program can pass to a chained program.

Syntax:

COMMON <*variable*>,<*variable*>

```
COMMON A$, COUNT, N
```

Explanation:

ST BASIC treats all COMMON statements in a program as one consecutive list of variables. A program can contain any number of COMMON statements.

COMMON statements can appear anywhere in a program. It is good practice to place them at the beginning of a program.

Use COMMON with CHAIN.

See: CHAIN

Example:

The following example chains to a program named EMPLOYEE and passes the variables VAL!, NAME$, and the array variable SCALE():

```
Ok 350 COMMON VAL! NAME$, SCALE()
Ok 360 CHAIN "EMPLOYEE"
```

CONT

The CONT command resumes program execution from the BREAK mode.

Syntax:

CONT

```
CONT
```

Explanation:

A BREAK, a STOP statement in a program, or [Control] [G] (unless trapped) puts ST BASIC in Break mode. In Break mode, you can use direct mode statements to change intermediate program values.

Use CONT to continue execution.

You can also use a direct mode GOTO statement to direct execution to a particular line in the program.

Example:

```
Ok 10 N=5
Ok 20 FOR X=1 TO 5
Ok 30 N=N-1
Ok 40 PRINT N
Ok 50 NEXT X
Ok RUN
 4
 3
 2
    [press [Control] [G]]
 -- Break -- at line 30
Br CONT
 1
 0
Ok
```

COS

The COS function returns the cosine of a number.

Syntax:

X = COS(<*numeric expression*>)

```
X=COS(Y)
```

Explanation:

The COS function returns a real number. The number is the cosine value of the angle in the numeric expression.

All ST BASIC trigonometric functions require that you specify angles in radians.

Example:

```
Ok 10 PI=3.14159
Ok 20 DEGREES=180
Ok 30 RADIANS=DEGREES*(PI/180)
Ok 40 ANS!=COS(RADIANS)
Ok 50 PRINT "THE COSINE IS "; ANS!
Ok RUN
THE COSINE IS -1
Ok
```

CVD
CVI
CVS

The CVD, CVI, and CVS functions convert byte strings to numeric variable types. Used to convert ASCII numbers read from random files.

Syntax:

CVD(<*8-byte string*>)
CVI(<*2-byte string*>)
CVS(<*4-byte string*>)

```
CVD(A$) A$=8-byte string
CVI(B$) B$=2-byte string
CVS(C$) C$=4-byte string
```

Explanation:

ST BASIC stores numbers in a random file as strings of bytes. To read the numbers from the file, the strings must be converted to the proper numeric data type. The functions do not change the

value of the number, only the data type. These strings are the exact byte representation of the stored numbers. They are not printable character strings.

The CVD function converts an 8-byte string into a real number.

The CVI function converts a 2-byte string into an integer.

The CVS function converts a 4-byte string to a real number.

If the string read from the file is shorter than the length required for conversion, it is padded to the right with binary zeroes.

The MKD$, MKI$, and MKS$ functions are the reverse of the CVD, CVI, and CVS functions.

Example:

```
Ok 10 OPEN "R",#1,"NUMBERS"
Ok 20 FIELD #1, 2 AS A$,4 AS B$,8 AS C$
Ok 30 GET #1,REC%
Ok 40 I%=CVI(A$)
Ok 50 X!=CVS(B$)
Ok 60 Y#=CVD(C$)
Ok 70 PRINT I%,X!,Y#
Ok 80 CLOSE #1
Ok 90 END
```

If you run this program, you will get one set of numbers from the file and print them.

DATA

The DATA statement defines a list of constants that a READ statement can assign to variables.

Syntax:

DATA <*constant*>,<*constant*>

```
DATA 25,15,925,word
```

Explanation:

DATA statements allow you to assign fixed values to variables. They are assigned according to their order in a DATA statement.

Every DATA constant must have a corresponding READ variable, and vice versa. The constants and variables match according to the order in which they are listed; the first DATA constant relates to the first READ variable, and so on.

DATA constants can be integers, real numbers, or strings, in any combination. The data types for the constants in the DATA list, however, must match the variables assigned them in a READ statement. Do not put quotation marks around strings in a DATA statement.

DATA statements can be as long as you like, but you cannot write other statements on the same line as a DATA statement.

Though every constant must have a corresponding variable, you do not need a READ statement for every DATA statement. You can have many DATA statements in a program and you can assign them variables in a single READ statement. In that case, they match first according to the constants' order in the program, then by their order within the lines.

The RESTORE statement points READ statements to DATA statement lines.

See: READ, RESTORE

Example:

```
Ok 10 READ X
Ok 20 DATA 33.3,5,ALLOW ROOM FOR GROWTH
Ok 30 PRINT X
Ok 40 READ X,Y$
Ok 50 PRINT X,Y$
Ok RUN
 33.3
 5            ALLOW ROOM FOR GROWTH
```

DEF FN

The DEF FN statement defines user specified functions.

Syntax:

DEF FN<*function name*>[(<*parameter,parameter*>)]=
<*definition*>

```
DEF FN(A)=A*2+5
```

Explanation:

DEF FN allows you to define your own ST BASIC function for use in a program. The name of the function can be any legal variable name.

The variable list in parentheses is optional. You can use any variable type except arrays. These variables are local to the function you define and do not affect variables of the same name elsewhere in the program. The variables in parentheses can be regarded as place holders for the values you pass to the function when you call it. The values you pass to your function must match those in parentheses in type and number.

You can use any global variables in your program within the function definition. They will be treated exactly as the function definition states. If you change their values within the function, they will take on their new values throughout the program.

The definition is an expression that defines what the function does. The description is limited to one program line. If the function name includes a type specification, such as FNA$, the definition may not conflict with that type. The parameters passed to the function (in parentheses) must also conform to that type.

Example:

```
Ok 10 INPUT "WIDTH OF MATERIAL IN INCHES";
MATERIAL.WIDTH
Ok 20 INPUT "WIDTH OF WINDOWSILL IN INCHES";
WINDOW.WIDTH
Ok 30 PANELS.NEEDED=WINDOW.WIDTH/MATERIAL.WIDTH
Ok 40 INPUT "LENGTH OF WINDOWSILL IN INCHES";
WINDOW.LENGTH
Ok 50 YARDAGE.NEEDED=PANELS.NEEDED*WINDOW.LENGTH
Ok 60 INPUT "PRICE OF MATERIAL PER YARD"; PRICE.YARD!
```

```
Ok 70 DEF FNSLACK=YARDAGE.NEEDED/15+YARDAGE.NEEDED
Ok 80 DEF FNCOST!=(PRICE.YARD!/36)*FNSLACK
Ok 90 PRINT "YOU NEED ";FNSLACK;" INCHES OF
";MATERIAL.WIDTH;" INCN MATERIAL.":PRINT
"YOUR COST IS: ";FNCOST!
Ok 100 DEF FNINYARDS=FNSLACK/36
Ok 110 PRINT FNSLACK; "INCNES IN YARDS IS ";FNINYARDS
Ok RUN
WIDTH OF MATERIAL IN INCHES? 30
WIDTH OF WINDOWSILL IN INCHES? 60
LENGTH OF WINDOWSILL IN INCHES? 60
PRICE OF MATERIAL PER YARD? 2.00
YOU NEED 128 INCHES OF 30 INCN MATERIAL.
YOUR COST IS 7.11111
 128 INCHES IN YARDS IS 3.55555
Ok
```

DEF SEG

The DEF SEG statement establishes the mode of operation of PEEK and POKE and the offset used by the commands.

Syntax:

DEF SEG [<*numeric expression*>]

```
DEF SEG 0
DEF SEG 1
```

Explanation:

The modes of operation are defined according to the following:

If DEF SEG > 0, then 1 byte is PEEKed or POKEd, and the value of the numeric expression used in DEF SEG is used as the offset of the address specified in PEEK and POKE.

If DEF SEG = 0, then 2 bytes are PEEKed or POKEd, and the value of the numeric expression used in DEF SEG is used as the offset of the address specified in PEEK and POKE.

If DEF SEG = 0 and the address is specified by DEFDBL, then 4 bytes (long integer) are PEEKed and POKEd.

Example 1:

```
10 DEF SEG=0
20 DEFDBL S:S=SYSTAB+20:'GRAPHICS BUFFER POINTER
30 X=PEEK(S):'X IS A 4 BYTE VALUE
40 RESET:'PUTS CURRENT SCREEN INTO GRAPHICS BUFFER
50 BSAVE "SCREEN",X,32767
60 CLEARW 2:'CLEAR SCREEN IMAGE
70 BLOAD "SCREEN",X:'RETURN SCREEN TO GRAPHICS BUFFER
80 OPENW 2:'TRANSFER GRAPHICS BUFFER TO WINDOW
```

Example 2:

```
10 DEF SEG=100
20 PRINT PEEK(500)
```

Note: Will print a 1-byte integer from absolute location 600.

Example 3:

```
10 DEF SEG=0
20 LOC#=175000
30 PRINT PEEK(LOC#)
```

Note: Will print a 4-byte long integer from location 175000.

DEFDBL

The DEFDBL statement declares a range of letters as defining real numbers.

Syntax:

DEFDBL <*letter*>-<*letter*>

```
DEFDBL A
DEFDBL A-D
```

Explanation:

The DEFDBL statement declares that the variables whose names start with any of the given letters are real numbers. You can use a single letter as a parameter or a range of letters, such as A-D.

Type declaration characters always overrule DEFDBL statements. DEFDBL statements can only be entered as the first statements in a program. DEFDBL is always used in conjunction with DEFSEG, PEEK, or POKE.

See: DEFSEG

Caution: DEFDBL statements alter the ST BASIC interpretation of program lines.

Example:

```
Ok 10 DEFDBL X-Y
Ok 20 X=123123412345123456
Ok 30 Y=&H333
Ok 40 PRINT X,Y
RUN
 1.23123392D+017 819
Ok
```

DEFINT

The DEFINT statement declares a range of letters as defining integers.

Syntax:

DEFINT <*letter*>-<*letter*>

```
DEFINT A
DEFINT A-D
```

Explanation:

The DEFINT statement declares that the variables whose names start with one of the given letters are integers. You can use one letter as a parameter or a range of letters, such as M-Z.

Type declaration characters overrule DEFINT statements.

Caution: DEFINT statements alter the ST BASIC interpretation of program lines. If you declare a variable as integer with a DEFINT statement, ST BASIC treats it as integer even if you erase the DEFINT statement.

Example:

```
Ok 10 DEFINT X-Y
Ok 20 X=78.9
Ok 30 Y=78.1
Ok 40 PRINT X,Y
Ok RUN
 78    78
Ok
```

DEFSNG

The DEFSNG statement declares a range of letters as defining real numbers.

Syntax:

DEFSNG *<letter>-<-letter>*

```
DEFSNG A
DEFSNG A-D
```

Explanation:

The DEFSNG statement defines the variable names that start with one of the given letters as real numbers. You can use one letter as a parameter or a range of letters, such as A-D.

Type declaration characters always overrule DEFSNG statements.

Caution: DEFSNG statements alter the ST BASIC interpretation of program lines.

Example:

```
Ok 10 DEFSNG X-Y
Ok 20 X=23D+16
Ok 30 Y=456654456654
Ok 40 PRINT X,Y
Ok RUN
 2.3E+17    4.56654E+11
```

DEFSTR

The DEFSTR statement declares a range of letters as defining strings.

Syntax:

DEFSTR <*letter*>-<*letter*>

```
DEFSTR A
DEFSTR A-D
```

Explanation:

The DEFSTR statement declares that all variables whose first letters are on the parameter list are strings. The parameters can be a single letter or a range of letters, such as M-Z.

A type declaration character always overrules a DEFSTR statement. The default type of variables is real numeric.

Caution: DEFSTR statements alter the ST BASIC interpretation of program lines. If you declare a variable as a string with a DEFSTR statement, ST BASIC treats it as real numeric even if you erase the DEFSTR statement.

Example:

```
Ok 10 DEFSTR A-C
Ok 20 A="12.7.42"
Ok 30 B="1066"
Ok 40 C="4.12.XX"
Ok 50 PRINT A,B,C
Ok RUN
12.7.42   1066   4.12.XX
Ok
```

DELETE

The DELETE command erases program lines from memory.

Syntax:

DELETE <*line number list*>

```
DELETE -40
DELETE 20
DELETE 20, 30
DELETE -30
```

Explanation:

DELETE erases the lines you specify. It is more efficient to delete a single line by typing the line number and pressing **[Return]**.

Example:

```
Ok 10 X=10
Ok 20 Z=20
Ok 30 PRINT X,Z
Ok DELETE 20-30
Ok LIST
   10 X=10
Ok
```

DIM

The DIM statement defines the number of dimensions and the number of elements in an array.

Syntax:

DIM <*array name*>(<*subscript*>,<*subscript*>)
(,<*array name*>[<*subscript*>])

```
DIM A$(5)
DIM X(5,10,4)
DIM B$(10),C$(20)
DIM X(5,10,4),Y(1,2,8)
```

Explanation:

The DIM statement reserves space for a string or numeric array by specifying the number of dimensions and the upper bound of elements in each. The number of dimensions depends upon the number of subscripts. One subscript means one dimension, two subscripts means two dimensions. The number of elements and dimensions you can specify is dependent upon available memory, but the maximum number of dimensions in any case is 15.

The lower bound of each dimension is 0 or 1, depending upon the OPTION BASE.

DIM automatically sets the initial value of the elements at zero or null.

In ST BASIC, arrays are dynamic. You can dimension the array with DIM, erase the array later in the program, and declare it again with DIM using the same name but with new dimensions. With dynamic arrays, you can also use a numeric variable to dimension the array.

You can use an array without declaring it first with a DIM statement. If you do, the array is declared automatically with a default upper bound of 10 elements in each dimension. For example, if the first reference to ARRAY A is

```
ARRAY A(7,3)
```

the array is set up as if it had been declared with

```
DIM A(10,10)
```

The default number of dimensions allowed is 4 for integers and 3 for strings and real numbers.

Note: ST BASIC allows one-third of the free memory to be declared as arrays. However, the total size of all arrays can't exceed 32K, regardless of the amount of free memory.

Example:

```
Ok 10 DIM HOUSES$(1,1,1)
Ok 20 HOUSES$(0,0,0)="FLOORPLAN1"
Ok 30 HOUSES$(0,0,1)="FL0ORPLAN3"
Ok 40 HOUSES$(0,1,0)="FLOORPLAN3"
Ok 50 HOUSES$(0,1,1)="FLOORPLAN3"
Ok 60 HOUSES$(1,0,0)="FLOORPLAN1"
Ok 70 HOUSES$(1,0,1)="FLOORPLAN2"
Ok 00 HOUSES$(1,1,1)="FLOORPLAN2"
Ok 90 IF HOUSES$(1,0,0)="FL0ORPLAN2" THEN GOTO 300
```

DIR

The DIR command lists the files on a disk.

Syntax:

DIR [<*disk drive:*>][<*filename, filetype*>]

```
DIR
DIR A:
DIR B:BAS.PRG
DIR B:*.PRG
DIR B:BAS.*
DIR B:*.*
DIR B:BAS.PR?
```

Explanation:

The DIR command displays the directory of the disk in the current drive.

You can specify which drive and what files you want displayed. The * and ? act as wild card designators.

* indicates a "don't care" specification for an arbitrary field, such as: *.BAS (for any file of type .BAS) or FIG.* (for any type file named FIG.) or B*.BAS (for any type.BAS file beginning with a B).

? acts as a single character "don't care" designator, such as: ?IG.BAS (for any file with a 3 letter name ending in IG.BAS. — FIG.BAS, PIG.BAS, BIG.BAS).

Example:

`Ok DIR`	Directory of all files on the current disk
`Ok DIR A:`	Directory of all files on Disk A
`Ok DIR B:BAS.PRG`	Checks for file BAS.PRG on Disk B
`Ok DIR B:*.PRG`	Directory of all type .PRG files on Disk B
`Ok DIR B:BAS.*`	Directory of all files named BAS of any type on Disk B
`Ok DIR B:*.*`	Directory of all files of any type on Disk B
`Ok DIR B:BAS.PR?`	Directory of files on Disk B beginning with BAS and with an extender beginning with PR.

EDIT

The EDIT command invokes the ST BASIC editor.

Syntax:

EDIT <*line number*> ED <*line number*>

```
EDIT      ED
EDIT 30   ED 30
```

Explanation:

The EDIT command invokes the ST BASIC editor. You can specify a line number to begin editing. If you don't, EDIT begins at the first line of the program currently in memory.

ELLIPSE

The ELLIPSE statement draws ellipses and elliptical arcs.

Syntax:

ELLIPSE <*horizontal center,vertical center,horizontal radius,vertical radius*>[<,*start angle,end angle*>]

```
ELLIPSE 50,80,100,50
ELLIPSE 50,80,100,50,900,1800
```

Explanation:

ELLIPSE draws an ellipse whose center is located at the point specified by the first two parameters: horizontal center and vertical center. The positions are in pixels starting from the upper left corner of the Output Window.

The third and fourth parameters, horizontal and vertical radii, are also expressed in pixels. The horizontal and vertical pixel count is dependent upon the resolution selected and the size of the Output Window.

The ellipse is drawn in the plot color (parameter 3 of the COLOR statement.)

The last two parameters, start angle and end angle, are optional. If they are not specified, ELLIPSE draws a full ellipse. If they are specified, ELLIPSE draws the part of an ellipse that lies between them. ELLIPSE draws an arc, not a solid colored pie-shaped segment.

Angles are expressed in degrees times 10. You would specify 45 degrees as 450, 180 degrees as 1800, etc. Zero degrees is to the right of the window, 90 degrees is toward the top, 180 degrees to the left, and 270 degrees at the bottom. ELLIPSE 100,80,40,50,0,3600 draws a full ellipse.

See: PELLIPSE, CIRCLE, PCIRCLE

Example:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 ELLIPSE 100,80,40,80
Ok 30 COLOR 1,0,2
Ok 40 ELLIPSE 100,80,40,80,300,900
Ok RUN
```
[Output Window will show black ellipse with 60 degree red arc at 30 degrees]
```
Ok
```

END

The END statement stops program execution, closes all files, and returns to command level.

Syntax:

END

```
END
```

Explanation:

You can put an END statement anywhere you want to return to command level. An END at the end of the program is optional.

END differs from STOP in that it closes all files, returns to command level, and does not produce a STOP message.

Example:

```
Ok 10 PRINT "THE PROGRAM"
Ok 20 PRINT "IS RUNNING"
Ok 30 PRINT "BUT WILL NEVER"
Ok 40 PRINT "REACH THE LAST"
Ok 50 PRINT "WORD OF THIS"
Ok 60 END
Ok 70 PRINT "PROGRAM"
Ok RUN
THE PROGRAM
IS RUNNING
BUT WILL NEVER
REACH THE LAST
WORD OF THIS
Ok
```

EOF

The EOF function returns true (-1) at the end of a sequential or random access file.

Syntax:

X = EOF(<*file number*>)

```
X=EOF(1)
```

Explanation:

When you write to a sequential file, its end is automatically marked. If you attempt to read past the end of a file, an error results. You can test whether you are at the end of a file with EOF.

EOF returns -1 if you are at the end of a file, 0 if not.

Example:

```
Ok 100 INPUT "FILE ";F$
Ok 110 IF LEN(F$)=0 THEN END
Ok 120 ON ERROR GOTO 20000
Ok 130 OPEN "I",1,F$
Ok 140 WHILE NOT EOF(1)
Ok 150 LINE INPUT #1,R$:?R$
Ok 160 WEND
Ok 200 ?:CLOSE 1:GOTO 100
Ok 20000 IF ERR=53 THEN ?"FILE ";F$;
"NOT FOUND":RESUME 100 ELSE ON ERROR GOTO 100
```

ERA

The ERA command deletes a file from the disk.

Syntax:

ERA [<*disk drive:*>]<*filename*>

```
ERA MYFILE.TXT
ERA B:MYFILE.TXT
```

Explanation:

The ERA command erases all files matching the filename from the drive specified. An erased file is not recoverable.

ERASE

The ERASE statement erases arrays.

Syntax:

ERASE <*array name*>,<*array name*>

```
ERASE A$, B$, C
```

Explanation:

ERASE erases an array so that you can redimension it or reclaim its memory space. You must erase arrays before you redimension them.

See: DIM

Example:

```
Ok 10 DIM PAYROLL$(10,10)
Ok 20 PAYROLL$(0,0)="BECKWITH, JOSEPHINE"
Ok 30 ERASE PAYROLL$
Ok 40 DIM PAYROLL$(5,5,5)
```

ERL
ERR

The ERL and ERR variables are reserved variables used in error handling subroutines.

Syntax:

X = ERL
X = ERR

```
X=ERL
X=ERR
```

Explanation:

ERL contains the line number where an error occurred. ERR contains the error code. ERL and ERR are reserved variables; you cannot write them on the left of the equal sign in an assignment statement.

If the statement or command in which the error occurred is in direct mode, the value of ERL is zero. If an error occurs in direct mode, the program always halts.

If the statement is in indirect mode, write IF statements as follows:

IF ERL = <*error line*> THEN <*executable statement*>
IF ERR = <*error code*> THEN <*executable statement*>

See: ERROR statement for details on error trapping and examples of ERL and ERR in an error trapping subroutine.

ERROR

The ERROR statement simulates an ST BASIC run time error and transfers control to an error trapping routine.

Syntax:

ERROR <*numeric expression*>

```
ERROR X
```

Explanation:

You can define errors and error messages in your programs with the ERROR statement. ERROR assigns an error code number to an error. The number must be an integer expression.

Every time the error occurs, the program refers to the error code number. If the error code corresponds to an ST BASIC error code, the ST BASIC error message prints. If an error trap that you have written is in effect, control passes to your error trap routine.

Two predefined variables are associated with the ERROR statement: ERL and ERR.

When an error occurs, ERR contains the error code constant. You can use it to write error messages. For example: IF ERR = 100 THEN PRINT "PLEASE CHECK THE NUMBER AND REENTER"

ERL contains the line number where the error happened.

If no user error trap is set, the message corresponding to the value in ERR is printed and the program halts. This occurs if an ERROR statement is executed in direct mode whether you set a trap or not.

If you set a trap, the program enters the error-trapping routine. You can use ERR and ERL as you would any numeric variable. To exit the error trap, use RESUME, whether you entered the trap because of a trappable ST BASIC error or an ERROR statement.

If the error code equals a predefined ST BASIC error code, the program simulates the error and prints the error message for that code.

When you define your own errors, it's a good idea to give your error codes values that are much greater than the ST BASIC codes. That way, you will not need to change your program even if the ST BASIC error codes are revised in the future.

See: ON ERROR, GOTO, RESUME

You can simulate errors in both direct and indirect mode. Here is an example in direct mode:

```
Ok ERROR 55
You cannot OPEN or KILL a file already open
```

The following example is in indirect mode:

```
  .
  .
  .
Ok 500 ON ERROR GOTO 550
Ok 510 INPUT "DO YOU WISH TO RECEIVE EARNED INCOME
CREDIT";E$
Ok 515 IF E$="NO" THEN GOTO 600
Ok 520 INPUT "IS THE AMOUNT LISTED ON LINE 33 LESS
THAN $10,000";X$
Ok 525 IF X$="NO" THEN ERROR 200
Ok 530 IF ERR=200 THEN
Ok 535 PRINT "YOU ARE INELIGIBLE FOR EARNED
INCOME CREDIT."
Ok 540 IF ERL=525 THEN GOTO 600
Ok 550 RESUME
  .
  .
  .
Ok RUN
DO YOU WISH TO RECEIVE EARNED INCOME CREDIT? YES
IS THE AMOUNT LISTED ON LINE 33 LESS THAN $10,000? NO
YOU ARE INELIGIBLE FOR EARNED INCOME CREDIT.
```

EXP

The EXP function returns the constant e raised to an exponent.

Syntax:

X = EXP(<*numeric expression*>)

```
X=EXP(Y)
```

Explanation:

The constant e is the base of natural logarithms, approximately equal to 2.7182. EXP returns a real number.

The numeric expression must evaluate to < = 43.6682.

Example:

```
Ok 10 X=EXP(3.254)
Ok 20 Y=EXP(8.97)
Ok 30 PRINT X,Y
Ok RUN
 25.8937    7863.59
Ok
```

FIELD

The FIELD statement allocates variable space in random file buffers.

Syntax:

FIELD #<*file number*>,<*field width*> AS <*string variable*>
<,*field width*> AS <*string variable*>

```
FIELD #1,8 AS X$,4 AS Y$,2 AS 5$
```

Explanation:

You must write a FIELD statement to transfer information between random file disks and random buffers. The FIELD statement only allocates variable space; it does not move data.

The file number is the number you gave the file when you opened it. The field width defines the number of bytes to give to the string variable. For example, FIELD #10, 20 AS X$, 30 AS Z$ allocates the first 20 bytes of space X$ and the next 30 bytes to Z$.

You cannot allocate more space than you created when you opened the file. The default record length is 128 bytes. For any file, you can write as many FIELD statements as you want.

Reallocating field space does not cancel the original mapping; rather, the two maps co-exist. For example, if you specify

```
FIELD #10,20 AS X$,40 AS Z$,10 AS Y$
```

and

```
FIELD #10,70 AS N$
```

the first 20 bytes of N$ are also in X$, the next 40 also in Z$, and the final 10 also in Y$.

Do not use INPUT or LET to input into a variable that was declared in a FIELD statement. If you do, the variable's pointer moves to string space instead of to the buffer.

Example:

```
Ok 100 OPEN "R",#5,"TAXES",40
0 110 FIELD #5,20 AS I$,10 AS D$,10 AS E$
```

FILL

The FILL statement fills shapes with colors or patterns.

Syntax:

FILL <*numeric X expression*>,<*numeric Y expression*>

```
FILL 150,80
```

Explanation:

Fills drawn shapes with shapes or patterns defined in a previous COLOR statement. The X and Y coordinates provide the starting position for FILL.

See: COLOR

Example:

```
10 COLOR 1,2,1
20 CIRCLE 150,80,80
30 FILL 150,80
40 COLOR 1,1,1,4,4
50 FILL 150,80
```

FIX

The FIX function truncates a real number to an integer.

Syntax:

X = FIX(number)

```
X=FIX(Y)
```

Explanation:

FIX does not round off numbers; it simply truncates any decimal part. The integer expression must be between -32768 and 32767.

See: CINT, INT

Example:

```
Ok 10 X=239.77
Ok 20 PRINT FIX(X)
Ok 30 PRINT FIX(-678.3)
Ok RUN
 239
-678
Ok
```

FLOAT

The FLOAT function converts an integer to a real number.

Syntax:

X = FLOAT(<*integer expression*>)

```
X=FLOAT(Y)
```

Explanation:

FLOAT does not change the appearance of the integer, but assigns it more room in memory. The integer expression must be between -32768 and 32767.

Example:

```
Ok 10 X=FL0AT(97)
Ok 20 PRINT X
Ok RUN
 97
```

FOLLOW

The FOLLOW command follows the values of program variables.

Syntax:

FOLLOW <*variable*>[,<*variable*>]

```
FOLLOW N
FOLLOW N, B
```

Explanation:

The FOLLOW command is a debugging tool that keeps track of all program variables. Each time the value of a specified variable changes, FOLLOW prints the variable name, its value, and the number of the program line on which it changed. The UNFOLLOW command stops FOLLOW.

Example:

```
Ok 10 FOR X=1 TO 3
Ok 20 N=N+1
Ok 30 B=B+1
Ok 40 PRINT N
Ok 50 PRINT B
Ok 60 NEXT X
Ok RUN
 1
 1
 2
 2
 3
 3
Ok FOLLOW N,B
Ok RUN
N! = 1 at line 20
B! = 1 at line 30
 1
 1
N!=2 at line 20
B!=2 at line 30
 2
 2
N!=3 at line 20
B!=3 at line 30
 3
 3
Ok UNFOLLOW
Ok
```

FOR

The FOR statement creates a loop that executes a given number of times.

Syntax:

FOR <*counter variable*> = <*numeric expression*> TO <*numeric expression*> [STEP<*numeric expression*>]

```
FOR I=1 TO 5 STEP 1
```

Explanation:

The FOR statement sets the starting and ending values of a counter variable and the value to be added to it each time the FOR . . . NEXT loop executes.

The value added to the counter variable is 1 unless otherwise specified by STEP. The STEP can be positive or negative.

The NEXT causes the instructions between FOR and NEXT to repeat if the value of the counter variable is not greater than the end value specified by TO. When the counter's absolute value is greater than the end absolute value, program execution passes to the line after NEXT.

You can nest FOR/NEXT statements. In other words, you can have a loop within a loop. When you nest loops, the NEXT statement for the inner loop must come before that of the outer loop.

See: NEXT

Example:

```
Ok 10 FOR X=1 TO 5
Ok 20 PRINT X
Ok 30 NEXT
Ok 40 PRINT "THE VALUE OF THE COUNTER VARIABLE IS"X
Ok RUN
 1
 2
 3
 4
 5
THE VALUE OF THE COUNTER VARIABLE IS 6
Ok
```

```
Ok 10 FOR X=2 TO 1 STEP -1
Ok 20 FOR Y=1 TO 5
Ok 30 PRINT X
Ok 40 PRINT Y
Ok 50 NEXT Y
Ok 60 NEXT X
Ok RUN
Ok
 2
 1
 2
 2
 2
 3
 2
 4
 2
 5
 1
 1
 1
 2
 1
 3
 1
 4
 1
 5
Ok
```

FRE

The FRE function returns the number of unused bytes in memory.

Syntax:

X = FRE(<*dummy argument*>)

```
X=FRE(0)
```

Explanation:

FRE requires a dummy argument. Use any argument to find the number of free bytes in memory.

Example:

```
Ok PRINT FRE(0)
 43000
```

Note: The size of ST BASIC arrays is limited to 32 Kilobytes regardless of the amount of free memory. The arrays must not exceed a third of the total size of free memory.

FULLW

The FULLW statement sets ST BASIC windows to full screen size.

Syntax:

FULLW <*numeric expression*>

```
FULLW 2
```

Explanation:

Sets the specified window to full screen. The windows are as follows:

0 = The Edit Window.
1 = The List Window.
2 = The Output Window.
3 = The Command Window.

Example:

```
Ok 10 FULLW 2:CLEARW 2
OK 20 PRINT "HELLO"
OK RUN
```

GEMSYS

The GEMSYS function allows the user to access the operating system's AES interface.

Syntax:

GEMSYS(<*AES Op Code*>)

```
GEMSYS(X)
```

Explanation:

The AES control arrays can be accessed through the GB structure, using the PEEK command.

Example:

```
 10 REM PRINT MOUSE X,Y POSITION AND BUTTON STATES
 20 A#=GB
 30 CONTROL=PEEK(A#)
 40 GLOBAL=PEEK(A#+4)
 50 GINTIN=PEEK(A#+8)
 60 GINTOUT=PEEK(A#+12)
 70 ADDRIN=PEEK(A#+16)
 80 ADDROUT=PEEK(A#+20)
 90 GEMSYS(79)
100 PRINT PEEK(GINTOUT+2)
110 PRINT PEEK(GINTOUT+4)
120 PRINT PEEK(GINTOUT+6)
130 PRINT PEEK(GINTOUT+8)
```

GET

The GET statement reads a record from a random disk file into a file buffer.

Syntax:

GET [#]<*file number*> [,<*record number*>]

```
GET #1,5
```

Explanation:

The file number is the number you gave the file when you opened it. The record number is optional. If you leave it out, the next record after the first GET or PUT goes into the buffer. The greatest record number you can have is 32767.

See: OPEN for an example of GET in context.

Example:

```
Ok 100 IF X$="YES" THEN GET#5,TYPE%:GOTO 200
```

GOSUB

The GOSUB statement passes program control to a subroutine.

Syntax:

GOSUB <*line number*> or GOSUB <*label name*>

```
GOSUB 250
GOSUB ENTRY
```

Explanation:

The GOSUB statement is paired with the RETURN statement, which passes control back to the program statement immediately following GOSUB.

The line number or symbolic label indicates the line on which the subroutine begins.

You can call a subroutine from another subroutine. Subroutines can't be nested more than 16 deep, however.

You can write more than one RETURN statement into your subroutine. If you are testing for conditions that determine a program's progress, you might have several RETURNS in a subroutine.

Note: It is advisable to use symbolic labels rather than line numbers with the GOSUB statement.

Example:

```
Ok 10 GOSUB 100
Ok 20 REM RETURN POINT OF SUBROUTINE
Ok 30 PRINT A
Ok 40 END
Ok 100 REM START OF SUBROUTINE
Ok 110 GOSUB BOO
Ok 120 A=5*5
Ok 130 RETURN
Ok 140 BOO: PRINT "BOO!"
Ok 150 RETURN
Ok RUN
 BOO!
 25
Ok
```

GOTO

The GOTO statement passes program control unconditionally to a given line number.

Syntax:

GOTO <*line number*> or GOTO <*label name*>

```
GOTO 50
GOTO ENTRY
```

Explanation:

The GOTO statement passes program control to a specified line and resumes execution from there. If you GOTO a nonexecutable statement, execution begins at the first executable statement after the specified statement.

Note: It is advisable to use symbolic labels rather than line numbers with the GOTO statement.

Example:

```
Ok 10 TOP: INPUT "PLEASE ENTER BENEFICIARY'S
NAME";NAME$
  .
  .
  .
Ok 100 INPUT "DO YOU WISH TO END THIS PROGRAM";ANSWER$
Ok 120 IF ANSWER$="YES" THEN GOTO 200
Ok 130 GOTO TOP
Ok 200 END
```

GOTOXY

The GOTOXY statement places output cursor at column and row position.

Syntax:

GOTOXY <*Column Position*>,<*Row Position*>

```
GOTOXY X,Y
```

Explanation:

GOTOXY places output cursor at the column and row position specified by the two parameters.

Example:

```
10 GOTOXY 2,3
20 PRINT "COLUMN2,ROW3"
```

HEX$

The HEX$ function returns a string that is the hexadecimal representation of a number.

Syntax:

X = HEX$(numeric expression)

```
X=HEX$(Y)
```

Explanation:

A hexadecimal number is a base 16 integer. Hexadecimal numbers are written using the digits 0 through 9 and the characters A through F to represent the values 1 through 15.

HEX$ does not add a leading &H to the hexadecimal number it returns. If you want to use the value in a program, you must prefix it with &H to establish that it is in hexadecimal notation.

HEX$ rounds real numbers to integers before evaluating them.

The normal legal range for integers is -32768 to 32767.

Attempting to assign an address expression to an integer variable leads to an integer overflow error, unless you assign the value to the variable using VAL (see following example).

Example:

```
Ok 10 A%=VAL("&H"+HEX$(FRE(0)))
Ok 20 PRINT A%
Ok RUN
-22536
Ok
```

IF

The IF statement sets conditions that determine program flow.

Syntax:

IF <*logical expression*> THEN <*statement*> <*:statement*>
[ELSE <*statement*> <*:statement*>]

```
IF X=Y THEN PRINT A:GOTO 250
ELSE GOTO 30
```

Explanation:

The IF statement evaluates an expression that is either true (not zero) or false (0). If the expression is true, the statements following THEN are executed. If false, execution continues at the statement after ELSE. If there is no ELSE, execution continues at the next executable line.

You can use IF statements within IF statements. Each ELSE matches with the nearest THEN. THEN or ELSE clauses are valid only within the context of an IF statement.

You can write a FOR or WHILE loop within the THEN or ELSE clause of an IF statement. The FOR or WHILE statement must be complete within the THEN or ELSE clause: the matching NEXT must be in the same clause as the FOR statement and the matching WEND must be in the same clause as the WHILE statement. See the first example below.

When you use an IF statement within a FOR or WHILE statement (all as part of the same statement line), the closing NEXT or WEND also closes the IF construct. See the second example below.

Example 1:

```
Ok 5 A%=5
Ok 10 IF A%>3 THEN FOR K%=1 TO
5:PRINT A%*K%:NEXT ELSE FOR
K%=1 TO 5:PRINT A%/K%:NEXT
Ok RUN
 5
 10
 15
 20
 25
Ok
```

Example 2:

```
Ok 10 FOR X=1 TO 5:IF X<3 THEN PRINT X*X:NEXT:PRINT
"DONE"
Ok RUN
 1
 4
DONE
```

(The NEXT is always executed)

INP

The INP function returns a byte value from a selected input port.

Syntax:

X = INP(<*port number*>)

```
X=INP(3)
```

Explanation:

The port number must be in the range 0 to 65535. The INP function is the complement of the OUT statement.

To read the port status, use a negative port value (INP (-3)). A 0 indicates no character available; -1 indicates a character available.

The following port assignments apply to the ATARI ST Computer:

0 = PRINTER (Parallel Port)
1 = AUX (RS-232)
2 = CONSOLE (Keyboard)
3 = MIDI (Musical Instrument Digital Interface)

Example:

```
Ok 200 Y=INP(3)
Ok 210 IF INP(3)>X THEN GOTO 200
```

INPUT

The INPUT statement lets you enter data,while the program is running and assigns the data to program variables.

Syntax:

INPUT [;] [<*prompt string*> <*; or,*>] <*variable*> ,<*variable*>

```
INPUT A$
INPUT "NAME: ",A$
INPUT "NAME";A$
INPUT X,Y,Z
INPUT "Height, Weight, Age", X,Y,Z
```

Explanation:

The INPUT statement prompts you for input during program execution and waits for your response. After you type a response, press [Return] to pass it to the program.

The prompt string is a string constant, and must be in quotes. The variables can be string or numeric. Your responses must match the type of the variables. String responses are not placed within quotes.

If you use a prompt string, the INPUT statement prints it on the screen as the prompt. The prompt string appears as a question or a statement, depending on whether you use a comma or a semicolon.

If you separate the prompt string from the variables with a semicolon, the INPUT statement adds a question mark and a space to the end of the prompt string.

If you separate the prompt string from the variables with a comma, the prompt prints without a question mark, and without a space after the last character in your prompt string. You type your response on the same line. For this reason, you need to include a space as the last character in your prompt string if you want a space between the prompt and your response.

If you do not write a prompt string, or if you write a null string, INPUT prints a question mark and a space and awaits your input.

The INPUT statement prints a prompt for each variable, and each response corresponds to an INPUT variable. If the number of variables and responses differ, an error occurs.

You must separate individual responses with commas. You can also use commas in your response if you enclose the response string in quotation marks.

You can enter one line of characters in response to an INPUT request. A carriage return or line-feed ends the line of input. The maximum line length is 255 characters.

Example:

```
Ok 10 INPUT "ENTER TODAY'S DATE: ",X$
Ok 20 INPUT "ENTER YOUR IDENTIFICATION NUMBER: ",Z$
Ok 30 IF Z$="359152" THEN GOTO 100
Ok 40 PRINT "ACCESS DENIED":END
Ok 100 PRINT "YOU'RE IN!":END
Ok RUN
ENTER TODAY'S DATE: 9 JULY 1983
ENTER YOUR IDENTIFICATION NUMBER: 359152
YOU'RE IN!
Ok
```

INPUT#

The INPUT# statement reads data from a sequential disk file to program variables.

Syntax:

INPUT#<*file number*>,<*variable*>,<*variable*>

```
INPUT#1,A$,X
```

Explanation:

The file number is the number you give the file when you open it. You assign the data in the file to variables. The types of a variable and its assigned data must match.

The INPUT# statement works much like the INPUT statement, except that it does not prompt. Before assigning the data item you enter to the variable, INPUT# skips any leading spaces, tabs, carriage returns, and line feeds you enter with the data. The first character that is not one of these is taken as the start of the data. A space, a carriage return, line feed, comma, or reaching 255 characters signals the end of the data.

There are three kinds of data for the INPUT# statement: numbers in any of the numeric formats, quoted strings, and unquoted strings.

Data is interpreted as a number if the variable you assign to it is numeric; otherwise it is taken as a string. Numbers are ended by reaching end-of-file or 255 characters, or by a line feed, carriage return, comma, or any character that is not a valid part of a number.

Strings are treated as quoted if the first non-space character is a quotation mark. Everything within a pair of quotation marks is taken as data in quoted strings. You cannot use a quotation mark as a character within the quoted string because the second quotation mark ends the string. Quoted strings are also ended by reaching end-of-file or 255 characters.

Unquoted strings can include quotation marks. They are ended by a carriage return, line feed, comma, reaching end-of-file or 255 characters. Trailing spaces in unquoted strings are ignored.

Example:

```
Ok 10 OPEN "I",#1,"BILLING"
Ok 20 INPUT#1,CUSTOMER$,INVOICE%,DATE$
```

INPUT$

The INPUT$ function returns a specified number of characters from the keyboard or a data file.

Syntax:

X$ = INPUT$(<*number of characters*>[,[#]<*file number*>])

```
X$=INPUT$(6)
X$=INPUT$(6,#1)
```

Explanation:

INPUT$ reads the specified number of characters from the keyboard or a file, and returns a string containing these characters. All characters are returned without translation, exactly as they are entered, without exception. For example, **[Control] [G]** from the terminal and **[Control] [Z]** from a data file are passed to the string.

If you input the string from a file, you must specify an open file number. If you attempt to read beyond the end of the file, an error results.

See: EOF

Example:

```
Ok 20 X$=INPUT$(6)
Ok 30 IF X$="GEORGE" THEN 1000 ELSE PRINT "WRONG":END
Ok 1000 PRINT "OK"
Ok RUN
ARNOLD
WRONG
Ok
```

INSTR

The INSTR function searches for one string within another and returns its position.

Syntax:

X = INSTR([<*starting point*>,] <*target string expression*>, <*pattern string*>)

```
X=INSTR(3,A$,"DO")
X=INSTR(3,A$,B$)
```

Explanation:

INSTR looks for the first occurrence of a pattern string within a target string and returns its position.

You can specify a starting point for the search. The optional starting point is an integer between 1 and 255.

The target string and pattern strings can be string constants, expressions, or variables.

INSTR returns 0 if the pattern string is longer than the target string, if the target string is a null string, or if the pattern string is not in the target string.

If the pattern string is null, INSTR returns a zero.

Example:

```
Ok 10 X$="HOW DO YOU DO?"
Ok 20 X=INSTR(3,X$,"DO")
Ok 30 PRINT X
Ok RUN
 5
Ok
```

INT

The INT function converts a number or expression to an integer.

Syntax:

X = INT(numeric expression)

```
X=INT(Y)
```

Explanation:

INT truncates decimal places.

Example:

```
Ok 10 X=INT(2.999)
Ok 20 PRINT X
Ok RUN
 2
Ok
```

KILL

The KILL statement deletes a disk file.

Syntax:

KILL<*string expression*>

```
KILL "FILE.DAT"
```

Explanation:

The string expression evaluates to a filename. KILL deletes the file associated with that filename. For example, KILL A$ deletes the file specified by A$. You can KILL any kind of disk file. You cannot KILL a file that is open at the time; an error occurs if you try.

The example creates a file named CALC.BAS. The file is then deleted by KILL.

Unlike ERA, KILL can be used within an ST BASIC program (i.e., 10 KILL "DATA.1").

Example:

```
Ok NEW
Ok 10 A=45:B=56
Ok 20 PRINT A+B
Ok 30 END
Ok SAVE CALC
Ok B$="CALC.BAS"
Ok KILL B$
Ok
```

LEFT$

The LEFT$ function returns a string that contains the leftmost characters of a string.

Syntax:

X$ = LEFT$(<*target string*> <*number of characters*>)

```
X$=LEFT$(A$,5)
```

Explanation:

LEFT$ starts at the leftmost character and returns as many consecutive characters as you specify. The number of characters must be a positive number between 1 and 255. Real expressions convert to integers.

The target string can be a string constant, variable, or expression.

If the number of characters is greater than the length of the target string, LEFT$ returns the entire target string. If the number of characters is zero, LEFT$ returns a null string.

Example:

```
Ok 10 INPUT "RADIUS";R
Ok 20 PRINT 3.1416*R^2
Ok 30 INPUT "ANOTHER AREA";C$
Ok 40 IF LEFT$(C$,1)="Y" THEN 10
Ok 50 END
RUN
.
.
.
RADIUS ?3
28.2735
ANOTHER AREA ?Y
RADIUS ?
```

LEN

The LEN function returns the length of a string.

Syntax:

Z = LEN(<*string expression*>)

```
X=LOC(1)
```

Explanation:

LEN returns the number of characters in a string as an integer. If the expression is a null string, LEN returns zero.

Example 1:

```
Ok 10 ADDRESS$="2114 PARKER ST, BIRDLAND, NEW YORK"
Ok 20 FOR X=1 TO LEN(ADDRESS$)
Ok 30 PRINT CHR$(42);
Ok 40 NEXT X
Ok RUN
**********************************
Ok
```

Example 2:

```
10 A$="THIS STRING IS 33 CHARACTERS LONG"
20 PRINT A$
30 PRINT LEN(A$)
RUN
THIS STRING IS 33 CHARACTERS LONG
33
```

LET

The LET statement assigns a value to a variable or array variable.

Syntax:

LET <*variable*> = <*expression*>

```
LET X=Y
LET X(1)=Y
```

Explanation:

Using LET to assign values to variables is optional. For example, LET X=Y and X=Y are identical in meaning. The variable and the expression can be strings or numbers. For numeric variables and expressions, the type of the expression converts to match the type of the variable.

Example:

```
Ok 10 LET NAME$="ALYSON"
Ok 20 TICKETOFFICE$="BATH, ENGLAND"
Ok 30 LET DESTINATION$="CANTERBURY"
Ok 40 DATE.OF.DEPARTURE=4.1
Ok 50 DATE.OF.ARRIVAL=4.8
Ok 60 LENGTH.OF.TRIP=DATE.OF.ARRIVAL -
DATE.OF.DEPARTURE
Ok 70 PRINT NAME$
Ok 80 PRINT TICKETOFFICE$
Ok 90 PRINT "DESTINATION: "DESTINATION$
Ok 100 PRINT "LENGTH OF TRIP: " LENGTH.OF.TRIP
Ok RUN
ALYSON
BATH, ENGLAND
DESTINATION: CANTERBURY
LENGTH OF TRIP: .7
Ok
```

LINE INPUT

The LINE INPUT statement requests input from the keyboard and assigns it to a string variable.

Syntax:

LINE INPUT[;] [<*prompt*>[,or ;]]<*string variable*>

```
INPUT LINE INPUT "NAME? ";A$
LINE INPUT;"NAME? ";A$
```

Explanation:

LINE INPUT is similar to the INPUT statement in that it asks you to enter data at the keyboard, but it accepts an entire line of up to 255 characters as a response. Your response is assigned to the string variable. A carriage return or line feed ends your input and sends it to the computer.

The optional prompt is a string that you write as an input request; LINE INPUT prints it in the Output Window and waits for your response. LINE INPUT does not automatically add a question mark or a space after the prompt, but you can write a question mark or space within the prompt string. Including a space is advisable, because otherwise your input will run together with the prompt, on the same line.

Example:

```
OK 10 LINE INPUT "REASON FOR RETURNING
MERCHANDISE";R$
OK 20 PRINT "THANK YOU. WE ARE PROCESSING YOUR
COMPLAINT"
Ok RUN
REASON FOR RETURNING MERCHANDISE?
WRONG SIZE, WRONG COLOR, TASTELESS STYLE.
THANK YOU. WE ARE PROCESSING YOUR COMPLAINT.
OK
```

LINE INPUT#

The LINE INPUT# statement requests input from a sequential disk file and assigns it to a string variable.

Syntax:

LINE INPUT#<*file number*>, <*string variable*>

```
LINE INPUT#1,A$
```

Explanation:

Like LINE INPUT, LINE INPUT# assigns a line of up to 254 characters as input to a string variable, but the input comes from a sequential disk file. The file number is the number you gave the file when you opened it.

LINE INPUT# reads all characters in a sequential file until it comes to a carriage return, and assigns them to the string variable. The next LINE INPUT# statement starts where the first left off, and assigns the next line, up to a carriage return, to the next string variable.

If a line feed immediately precedes a carriage return, they are treated as regular characters and do not end the line.

Example:

```
Ok 10 OPEN "O",#4,"SCORES"
Ok 20 LINE INPUT "GIVE TEAMS, WINNERS, AND
SCORES.",S$
Ok 30 PRINT#4,S$
Ok 40 CLOSE #4
Ok 50 OPEN "I",#4,"SCORES"
Ok 60 LINE INPUT#4,S$
Ok 70 PRINT S$
Ok 80 CLOSE #4
Ok RUN
GIVE TEAMS, WINNERS, AND SCORES.
USC & UCLA: USC. 50-3; CPSLO & FRESNO: CPSLO. 33-20
USC & UCLA: USC. 50-3; SPSLO & FRESNO: SPSLO. 33-20
Ok
```

LINEF

The LINEF statement draws a line.

Syntax:

LINEF <*point pair, point pair*>

```
LINEF 30,50,90,100
```

Explanation:

LINEF draws a line between the two point pair coordinates specified. The points are pixel positions counted from the upper left corner of the Output Window (0,0). The number of points available horizontally and vertically is dependent upon the system resolution chosen.

Example:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 LINEF 50,50,80,80
Ok RUN
```

[Output Window will show line drawn between two coordinate positions]

```
Ok
```

LIST

The LIST command displays program lines in the List Window.

Syntax:

LIST [<*line descriptor list*>]

```
LIST
LIST 10-50
LIST 10,30,50
LIST 10-30,70-90
LIST -30
```

Explanation:

LIST displays specified lines of the current program in the List Window.

LIST displays the entire program from beginning to end.

LIST 10 displays the single line number 10 of the program.

LIST 10-50 displays lines 10 through 50 of the program.

LIST 10, 30, 50 displays lines 10, 30 and 50 of the program.

LIST 10-30, 70-90 lists two groups of lines from 10 through 30 and 70 through 90.

LIST - 30 lists all lines up to line 30.

Pressing **[Control] [G]** stops LIST and returns to the Command Window.

LLIST

The LLIST command lists the program to your printer.

Syntax:

LLIST [<*line descriptor list*>]

```
LLIST
LLIST 10-50
LLIST 10,30,50
LLIST 10-30,70-90
LLIST -30
```

Explanation:

LLIST works the same way as LIST, but prints the specified lines on your printer.

The WIDTH LPRINT command sets the line width for your printer. ST BASIC sets line width to 72 characters. WIDTH LPRINT 40 would set it to 40 characters.

If a printer is not connected when the LLIST command is executed, ST BASIC will time out.

LOAD

The LOAD command LOADs program files.

Syntax:

LOAD <*filename*>

```
LOAD MYPROG
```

Explanation:

LOAD brings ST BASIC program files into memory. LOAD assumes a .BAS extender unless you specify otherwise. When you LOAD a program, any current program and its variables are cleared from memory.

Same as: OLD

LOC

The LOC function returns either a record number or the number of bytes read from or written to a file.

Syntax:

X = LOC(<*file number*>)

```
X=LOC(1)
```

Explanation:

When used after a GET or PUT to a random disk file, LOC returns the number of the record most recently read or written with GET or PUT. For example:

```
GET #1
PUT #1,LOC(1)
```

replaces record #1 in the slot from which it is read.

Used with sequential files, LOC returns the number of bytes read or written since the file was opened.

Example:

```
Ok 10 OPEN "R",#8,"FILE"
Ok 20 FIELD #8,20 AS Z$,3 AS V$
Ok 30 GET #8,C%
Ok 40 IF LOC(8)>25 THEN GOTO 90
```

LOF

The LOF function returns the number of bytes in the file.

Syntax:

X = LOF(<*file number*>)

```
X=LOF(1)
```

Explanation:

For a file just opened for output, the number of bytes is zero.

Example:

```
Ok 100 X=LOF(#5)
110 IF X>100 THEN PRINT "OPEN NEW FILE":
GOTO 200
```

LOG

The LOG function returns the natural logarithm of a number.

Syntax:
X = LOG(<*numeric expression*>)

```
X=LOG(N)
```

Explanation:
The numeric expression must be greater than zero.

Example:

```
Ok 10 PRINT LOG(23)/LOG(2)
Ok RUN
4.52356
Ok
```

LOG10

The LOG10 function returns the base 10 logarithm of a number.

Syntax:
X = LOG10(<*numeric expression*>)

```
X=LOG10(Y)
```

Explanation:
The numeric expression must be greater than zero.

Example:

```
Ok 10 X=LOG10(1000)
Ok 20 PRINT X
Ok RUN
 3
```

LPOS

The LPOS function returns the position of the line printer print head within the line printer buffer.

Syntax:

LPOS(X)

```
LPOS(X)
```

Explanation:

The position returned is the number of the characters printed since the last carriage return character. The backspace counts as -1. If you have printer control characters that alter the position of the print head, LPOS will not reflect the true position of the print head.

Example:

```
Ok 10 X=90
Ok 20 IF LPOS(X)>45 THEN GOTO 100
```

LPRINT

The LPRINT statement directs output to a printer.

Syntax:

LPRINT [<*list of expressions*>]
LPRINT USING <*format string expression*>;<*list of expressions*>

```
LPRINT A$;"=";X
LPRINT USING F$;A$,X
```

Explanation:

The LPRINT statement works like the PRINT and PRINT USING statements in this section, except that output goes to a line printer. You can set the assumed width of the line printer with the WIDTH LPRINT statement. Initially, it is 72 characters. The format string expression must be separated from the variable list with a semicolon. The listed expressions must be separated by commas.

See: WIDTH, LPRINT

Example:

```
Ok 10 LPRINT "THIS PRINTS ON THE PRINTER"
```

LSET

The LSET statement moves a string into a specified string variable without reassigning the string variable.

Syntax:

LSET<*string variable*> = <*string expression*>

```
LSET A$=B$
```

Explanation:

LSET is commonly used to move data to file buffers by LSETing into variables mapped into file buffers by a previous FIELD statement. LSET is not limited to this use, however.

If the string expression takes fewer bytes than you assigned to the string variable in a FIELD statement, LSET justifies the left margin and pads the string to the right with spaces.

If the string is longer than the destination, LSET ignores the extra characters.

If a string takes more bytes than you assigned it in the FIELD statement, characters to the right are dropped.

You must convert numbers and numeric variables to strings with MKD$, MKI$, or MKS$ before you LSET them.

The counterpart of LSET is RSET.

Example:

```
Ok 10 OPEN "I",#2,"TEST",5
Ok 20 FIELD #2,5 AS S$
Ok 30 LSET N$=NN$
```

MERGE

The MERGE command inserts a ST BASIC disk file into a program in memory.

Syntax:

MERGE <*filename*>

```
MERGE MYPROG
```

Explanation:

The MERGE command assumes a .BAS extender unless otherwise specified and inserts a file on disk into a file already in memory. As long as the line numbers of the two files are different, MERGE does not erase the original file. If any line numbers in the disk file duplicate line numbers in the file in memory, the disk lines replace the memory lines.

See: CHAIN

Example:

```
Ok 10 PRINT "THIS IS THE ORIGINAL PROGRAM"
Ok 20 PRINT "THIS LINE WILL BE DELETED BY
      THE MERGE"
Ok 30 PRINT "THIS LINE STAYS BECAUSE IT HAS A
      UNIQUE LINE NUMBER"
Ok SAVE ORIGINAL
Ok NEW
Ok 15 PRINT "THIS IS THE OVERLAY"
Ok 20 PRINT "THIS LINE REPLACES LINE 20 IN THE
      ORIGINAL PROGRAM"
Ok SAVE OVERLAY
Ok LOAD ORIGINAL
Ok MERGE OVERLAY
Ok RUN
THIS IS THE ORIGINAL PROGRAM
THIS IS THE OVERLAY
THIS LINE REPLACES LINE 20 IN THE ORIGINAL
PROGRAM
THIS LINE STAYS BECAUSE IT HAS A UNIQUE LINE NUMBER
Ok
```

MID$

The MID$ function returns a segment of a string.

Syntax:

X$ = MID$(<*string expression*>, <*starting point*>,[<*length*>])

```
X$=MID$(A$,5,10)
MID$(A$,5,5)="HELLO"
```

STATEMENT:

Assigns a value to a string segment.

Explanation:

MID$ returns a segment of a string. The starting point is a numeric expression pointing to the beginning of the segment. The length is a numeric expression specifying the length of the segment to the right of the starting point. If you omit the length parameter, MID$ returns all the characters after the starting point.

If the starting number is greater than the string length, MID$ returns a null string.

If the length of the segment is greater than the number of characters to the right of the starting point, all the characters after the starting point are returned.

MID$ can also be used to define a string segment.

See: RIGHT$, LEFT$

Example:

```
Ok 10 X$="MR. JAMES GRAHAM SCOTT"
Ok 20 Y$=MID$(X$,18,5)
Ok 30 PRINT Y$
Ok RUN
SCOTT
Ok
```

MKD$ MKI$ MKS$

The MID$, MKI$, and MKS$ functions convert ASCII strings representing numbers to byte strings for use in random file buffers.

Syntax:

X$ = MKD$(<*numeric expression*>)
X$ = MKI$(<*integer*>)
X$ = MKS$(<*numeric expression*>)

```
X$=MKD$(A)
X$=MKI$(B)
X$=MKS$(C)
```

A is a numeric value.
B is an integer value.
C is a numeric value.

Explanation:

MKI$ returns a 2-byte string. MKS$ returns a 4-byte string. MKD$ returns an 8-byte string.

You must convert ASCII numbers to strings with these functions before you can move them into a random file buffer with RSET or LSET. The CVD, CVI, and CVS functions are the reverse of the MKD$, MKI$, and MKS$ functions.

Example:

```
Ok 100 FINAL=(100/X)*(100-Y)
Ok 110 FIELD #2,5 AS Z$,5 AS B$
Ok 120 LSET Z$=MKI$(FINAL)
Ok 130 LSET B$=T$
Ok 140 PUT # 2
```

.
.
.

NAME

The NAME statement renames a file.

Syntax:

NAME <*old string expression*> AS <*new string expression*>

```
NAME "AUG.DAT" AS "LAST.DAT"
```

Explanation:

NAME simply gives a new name to a file that already exists. NAME does not alter the file or disk space in any way. Be sure the old file exists and the new name does not; otherwise, an error occurs.

Example:

```
Ok NAME "VERSION2.BAS" AS "FINAL.BAS"
```

NEW

The NEW command clears a file from memory, and optionally names the new program.

Syntax:

NEW [NAME]

```
NEW NEWPROG.BAS
```

Explanation:

Use NEW in preparation for writing a new program. If you have not saved the current file, you will lose it. If you use the NAME option, you can use the SAVE command later without a name.

Example:

```
Ok 10 X=SQR(25)
Ok 20 PRINT X
Ok NEW
Ok LIST
Ok
```

NEXT

The NEXT statement marks the end of a FOR/NEXT loop.

Syntax:

NEXT [<*counter*>] ,counter

```
NEXT X
NEXT X,Y
```

Explanation:

The NEXT statement in a FOR/NEXT loop sends program control to the beginning of the loop. The loop runs again if the counter variable is not greater than the limit set in a FOR statement.

Supplying the name of the counter variable is optional. The NEXT statement assumes the nearest counter variable.

If you have nested loops, you must specify which counter variable you are returning to at the end of the loop's execution. Use NEXT to direct execution first to the nested loop, then to the outer loop, by specifying first the nested counter variable, then the outer.

See: FOR

Example:

```
Ok 10 FOR Z=1 TO 3
Ok 20 PRINT "Y"
Ok 30 FOR Q=1 TO 2
Ok 40 PRINT "X"
Ok 50 NEXT Q,Z
Ok RUN
Y
X
Y
X
X
Y
X
X
Ok
```

OCT$

The OCT$ function returns the string expression of an octal (base 8) number.

Syntax:

X$ = OCT$(<*numeric expression*>)

```
X$=OCT$(Y)
```

Explanation:

OCT$ returns a string that is the base 8 equivalent of a decimal or hexadecimal value. The value of the decimal or hexadecimal expression is rounded to an integer before conversion. It must be between -32768 and 32767.

See: HEX$, STR$

Example:

```
Ok 10 X$=OCT$(3.4)
Ok 20 PRINT X$
Ok RUN
3
```

OLD

The OLD command loads an existing program file into memory.

Syntax:

OLD <*filename*>

```
OLD TEST
```

Explanation:

OLD closes all open files and erases any variables or data in memory before loading the named file from disk. Any ST BASIC program in memory is cleared by OLD.

The filename is the name you gave the file when you saved it. You need not include the default file type .BAS.

Same as: LOAD

Example:

```
Ok OLD TEST
Ok
The program TEST.BAS is now in program memory.
```

ON

The ON statement transfers program control to one of a list of program lines depending on the computed result of the numeric expression. The ON statement has two forms.

Syntax:

ON <*numeric expression*> GOTO <*line descriptor*> [<*line descriptor*>]
ON <*numeric expression*> GOSUB <*label*> [,<*label*>]

```
ON X GOTO INIT, 100, ENTRY, DONE
ON X GOSUB INIT, 100, ENTRY, DONE
```

Explanation:

The value of the numeric expression determines where program execution transfers. If the expression evaluates to 1, ON branches to the first label. If it evaluates to 2, ON branches to the second label, and so on.

Test the value before writing an ON statement.

Non-integer values round to the nearest whole number.

In the ON GOSUB statement, each numeric expression must be the number of the first line of a subroutine. The RETURN statement in the subroutine returns control to the first executable statement following the ON statement.

You can use any valid line descriptor in an ON statement, and you can write an ON statement anywhere in your program.

```
10 ON X GOTO 200,PAINT,400
```

If the value of X is 1 the program will jump to the line 200; if it is 2 it will jump to the statement labeled PAINT.

Example:

```
Ok 10 X=1
Ok 20 ON X GOTO 70,80,90,990
Ok 70 PRINT "SEASON TO DATE:"X + 1
Ok 80 PRINT "SEASON TO DATE:"X + 2
Ok 90 PRINT "SEASON TO DATE:"X + 3
Ok 120 X=X+1:GOTO 20
Ok 990 END
Ok RUN
SEASON TO DATE: 2
SEASON TO DATE: 3
SEASON TO DATE: 4
SEASON TO DATE: 4
SEASON TO DATE: 5
SEASON TO DATE: 6
Ok
```

ON ERROR GOTO

The ON ERROR GOTO statement provides a mechanism to detect run time errors and pass control to a line number when an error occurs.

Syntax:

ON ERROR GOTO <*line descriptor*>

```
ON ERROR GOTO 200
```

Explanation:

ON ERROR GOTO lets you handle run time errors by jumping to a given line number when ST BASIC detects an error. A line number, not a label, must be used as a parameter.

You can disable error handling, or restore ST BASIC's own error handling in an error routine, by using ON ERROR GOTO 0.

When you use ON ERROR GOTO 0 in an error trapping routine, ST BASIC prints its original error message. It is a good practice to always use ON ERROR GOTO 0 in an error trapping routine so that you can trap unexpected errors.

See: RESUME

Example:

```
Ok 80 ON ERROR GOTO 100
```

OPEN

The OPEN statement lets you input and output to a file or device.

Syntax:

OPEN <*mode*>,[#]<*file number*>,<*filename*>
[,<*record length*>]

```
OPEN "O",#1,"FILE.DAT",128
OPEN "I",#1,"FILE.DAT",128
OPEN "R",#1,"FILE.DAT",128
```

Explanation:

You must OPEN a disk file before you can move data into or out of it. The OPEN statement assigns the file an I/O buffer and determines the mode under which the file is accessible to I/O.

The file number is an integer expression with a value between 1 and 15. A file number belongs to a file for as long as it is open. Closing a file frees its number for reassignment. The record length is an integer expression that sets the record length for random files. It is optional. The default length is 128 bytes. A record length given for a sequential file is ignored.

The file mode is either sequential output or sequential input, or random input and output. Specify the mode with one of the following initials:

O Output for sequential files
I Input for sequential files
R Input and output for random files

These letters must be uppercase.

When you enter/input random access records, the first record number *must* be entered as **1** and all following record numbers must be sequential. That is, the first record is **1**, the second record is **2**, the third record is **3**, and so on. This can be done with a FOR, . . . NEXT loop. Records entered out of order cause the program to error out. Once the file is established, the records can be called *(GET* #1,VAR) in any order.

Example:

```
Ok 10 OPEN "R",#1,"FUNDS"
Ok 20 FIELD #1,10 AS V$,10 AS X$,30 AS N$
Ok 30 INPUT "ENTER A 4-DIGIT CODE",CODE!
Ok 40 GET #1,CODE!
```

OPENW

The OPENW statement opens one ST BASIC window.

Syntax:

OPENW <*window number*>

```
OPENW 2
```

Explanation:

Used to open one ST BASIC window that was previously closed using the CLOSEW command. The window opened will be the top one on the screen. If the window has already been opened, the window will remain the top window on the screen. <*Window number*> specifies the ST BASIC windows as follows:

0 - The Edit Window.
1 - The List Window.
2 - The Output Window.
3 - The Command Window.

Note: OPENW does certain bookkeeping chores internal to the ST BASIC interpreter that allow the system to keep track of the window status. Therefore, do not open ST BASIC windows (that were closed using CLOSEW) using direct calls to AES.

OPTION BASE

The OPTION BASE statement sets the base for array dimensions.

Syntax:

OPTION BASE *<1 or 0>*

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

Explanation:

You use OPTION BASE to set the minimum value for array subscripts within a dimension. The default base is zero; thus the first element in an array has a subscript of zero. You can set the array dimensions so they begin at 1 or reset them to zero.

You can use OPTION BASE as many times as required.

See: DIM

Example:

```
Ok 10 OPTION BASE 1
Ok 20 DIM A%(10)
Ok 30 OPTION BASE 0
Ok 40 DIM B%(10)
```

A% now has 10 elements, 1-10. B% has 11 elements, 0-10.

OUT

The OUT statement sends a byte to an output port.

Syntax:

OUT <*integer expression*>,<*integer expression*>

```
OUT 2,X
```

Explanation:

The first integer expression is the port number. The second expression is the byte you are sending to the port; it must evaluate to an integer between 0 and 255.

ATARI ST Computer ports are assigned as follows:

0 = PRINTER (Parallel Port)
1 = AUX (RS-232)
2 = CONSOLE (Keyboard)
3 = MIDI (Musical Instrument Digital Interface)

Example:

```
Ok 100 IF X%>5 THEN OUT 9,(X-2)
```

PCIRCLE

The PCIRCLE statement draws solid circles and pie shapes.

Syntax:

PCIRCLE <*horizontal center,vertical center,radius*>
[<,*start angle,end angle*>]

```
PCIRCLE 50,80,50
PCIRCLE 50,80,50,900,1800
```

Explanation:

PCIRCLE draws a solid color or patterned circle whose center is located at the point specified by the first two parameters: horizontal center and vertical center. The positions are in pixels starting from the upper left corner of the Output Window.

The third parameter, radius, is also expressed in pixels. The horizontal and vertical pixel count is dependent upon the resolution selected. The circle is drawn in the FILL color (parameter 2 of the COLOR statement.)

The last two parameters, start angle and end angle, are optional. If they are not specified, PCIRCLE draws a circle. If they are specified, PCIRCLE draws the part of a circle that lies between them. PCIRCLE draws a solid colored pie shaped segment, not an arc. Angles are expressed in degrees times 10. You would specify 45 degrees as 450, 180 degrees as 1800, and so on. Zero degrees is to the right of the window, 90 degrees is towards the top, 180 degrees to the left, and 270 degrees at the bottom. COLOR 1,3,1:PCIRCLE 100,30,30,0,3600 draws a solid green circle.

See: CIRCLE, ELLIPSE, PELLIPSE

Example:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 CIRCLE 100,50,40
Ok 30 COLOR 1,2,1
Ok 40 PCIRCLE 100,50,40,300,900
Ok RUN
```

[Output Window will show black circle with 60 degree red wedge at 30 degrees]

```
Ok
```

PEEK

The PEEK function returns the content of a memory location.

Syntax:

X = PEEK(<*memory location*>)

```
X=PEEK(Y)
```

Explanation:

PEEK returns the value at the specified memory location. The type of value returned is dependent upon the last DEF SEG statement as follows:

If DEF SEG > 0, PEEK returns a byte regardless of how the location to PEEK is specified. The location specified in PEEK will be offset by the value specified in the last DEF SEG statement.

If DEF SEG = 0,.PEEK returns a 2-byte word if location to PEEK is specified as a FLOAT expression.

IF DEF SEG = 0 and the address is specified by DEFDBL, PEEK returns a 4-byte long integer.

You must specify the memory address using a variable, as in the following example, rather than a constant.

See: POKE, DEF SEG

Note: When PEEKing, the 520ST Computer is switched into supervisory mode, meaning that you can access any location in memory including protected memory.

Example:

```
Ok 100 BYTE%=PEEK(234)
```

PELLIPSE

The PELLIPSE statement draws SOLID ellipses and elliptical pie shapes.

Syntax:

PELLIPSE <*horizontal center,vertical center,horizontal radius,vertical radius*>[<,*start angle,end angle*>]

```
PELLIPSE 50,80,100,50
PELLIPSE 50,80,100,50,900,1800
```

Explanation:

PELLIPSE draws an ellipse whose center is located at the point specified by the first two parameters: horizontal center and vertical center. The positions are in pixels starting from the upper left corner of the output window. The third and fourth parameters, horizontal and vertical radii, are also expressed in pixels. The horizontal and vertical pixel count depends upon the resolution selected. The ellipse is drawn in the FILL color (parameter 2 of the COLOR statement.)

The last two parameters, start angle and end angle, are optional. If they are not specified, PELLIPSE draws a full ellipse. If they are specified, PELLIPSE draws the part of an ellipse that lies between them. PELLIPSE draws a solid colored pie-shaped segment.

Angles are expressed in degrees times 10. You would specify 45 degrees as 450, 180 degrees as 1800 and so on. Zero degrees is to the right of the window, 90 degrees is towards the top, 180 degrees to the left, and 270 degrees at the bottom. COLOR 1,3,1:PELLIPSE 100,50,50,50,0,3600 draws a solid green ellipse.

See: ELLIPSE, CIRCLE, PCIRCLE

Example:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 ELLIPSE 100,80,40,80
Ok 30 COLOR 1,2,1
Ok 40 PELLIPSE 100,80,40,80,300,900
Ok RUN
```

[Output Window will show black ellipse with 60 degree red wedge at 30 degrees]

```
Ok
```

POKE

The POKE statement writes data to POKE to the memory.

Syntax:

POKE<*location to poke*>,<*data to poke*>

```
POKE 1565,X
```

Explanation:

POKE stores a value of the data to POKE in a memory location. The location to POKE is an absolute address given as a numeric expression. The data type is defined by the last previous DEF SEG statement and the manner in which the location to POKE is specified.

If DEF SEG > 0, data is a byte regardless of how location to POKE is specified. The location specified in POKE will be offset by the value specified in the last DEF SEG statement.

If DEF SEG = 0, data is 2-byte word if location to POKE is specified as a FLOAT expression.

If DEF SEG = 0 and address is specified by DEFDBL, data is a 4-byte long integer.

If the data expression evaluates outside the range 0 to 255, POKE stores the low-order byte of the result. For example:

```
Ok 5 DEF SEG=300000
Ok 10 POKE X%,257
```

has the same effect as

```
Ok 5 DEF SEG=300000
Ok 10 POKE X%,1
```

The complement of POKE is PEEK. You can use PEEK and POKE for passing arguments and data to machine language subroutines.

See: PEEK, DEF SEG

Example:

```
Ok 100 FOR LOC%=1 TO LEN(OUT,MSG$)
OK 120 POKE
MSG.LOC%+LOC%,ASC(MID$(OUT,MSG$,LOC$,))
OK 130 NEXT LOC%
```

Note: While POKEing or PEEKing, the computer is switched into supervisory mode, where you can access any location in the memory including protected memory. The system will crash if you POKE locations used by the TOS Operating System. Reboot the system if a crash occurs.

POS

The POS function returns the current position of the cursor on the screen or printer.

Syntax:

X = POS(<*dummy argument*>)

```
X=POS(0)
```

Explanation:

The leftmost position of the cursor is zero. POS does not necessarily give the physical position of the print head.

See: LPOS

Example:

```
Ok 40 X=POS(0)
Ok 50 PRINT "THE PRINT HEAD IS AT COLUMN: ";X
Ok 60 IF WIDTH.LINE<POS(0) THEN WIDTH.CHR=X
```

PRINT

The PRINT statement prints data to the Output Window.

Syntax:

PRINT [<*expression*><, *or* ;><*expression*>[<, *or* ;>]]

```
PRINT X,Y
PRINT X;Y
PRINT A$
?A$
```

Explanation:

PRINT sends expressions to the output window. You can use any number of expressions with the PRINT statement, separated by a comma or semicolon.

The punctuation used to separate the expressions determines the position of the expressions on the screen. ST BASIC divides a line into print zones consisting of 14 spaces each. When you use a comma to separate the expressions in the PRINT statement, ST BASIC prints each expression in the next available print zone. If you use a semicolon, ST BASIC prints string expressions consecutively, with no spaces separating them. Numeric expressions are printed together, with a space for the sign.

If you end a list of expressions with a comma, ST BASIC spaces to the next print zone, but does not move to a new line. If you end a list with a semicolon, ST BASIC leaves the cursor at the end of the last expression.

A question mark ? can be used in ST BASIC programs in place of PRINT. ? A means the same as PRINT A.

Example:

```
Ok 10 PRINT "TESTING ST BASIC"
Ok 20 PRINT
Ok 30 A$="ONE":B$="TWO":C$="THREE"
Ok 40 A=23:B=567:C=5
Ok 50 PRINT A$,B$,C$
Ok 60 PRINT A$;B$;C$
Ok 70 PRINT A,B,C
Ok 80 PRINT A;B;C;
Ok 90 END
Ok RUN
TESTING ST BASIC

ONE    TWO    THREE
ONETWOTHREE
 23    567    5
 23 567 5
Ok
```

PRINT#

The PRINT# statement outputs data to a disk file.

Syntax:

PRINT# <*file number*>,<*expression*>,<*expression*>

```
PRINT# 1,A$,X
?#
```

Explanation:

The PRINT# statement writes expressions to the file specified by the file number. The file number is the number you gave the file when you opened it. Each PRINT# statement creates a single record. Each expression used in the PRINT# statement creates a single field.

You can use any number of expressions with the PRINT# statement and separate each one with a comma or semicolon.

PRINT# writes the data to the file exactly as it would print on the screen using the PRINT statement.

You must express exactly how you want the data to appear on disk by punctuating it properly.

For example:

```
X$="Lewis"
Z$="C. S."
```

and you want to write

```
Lewis,C.S.
```

to disk. Since neither variable contains a comma, either before "Lewis" or after "C.S", the statement

```
Ok PRINT#1,X$;Z$
```

writes the data to disk as

```
Lewis C.S.
```

If you want to insert a comma as a delimiter, you must use the statement

```
Ok PRINT#1,X$;",";Z$
```

with the comma as a literal string in quotation marks.

Example:

```
Ok 50 PRINT#FIVE.TEX; A$,B$,C$
```

PRINT USING

The PRINT USING statement prints output according to a format.

Syntax:

PRINT USING<*string expression*>;<*list of expression''*>;
PRINT#<*file number*>,USING<*"string expression''*>;
<*list of expressions*>

```
PRINT USING FORM$;X,Y,Z
PRINT# 1,USING FORM$;X,Y,Z
?USING
```

Explanation:

The PRINT USING statement prints the data on the screen. The PRINT# USING statement prints the data on a disk file. You can print strings or numbers with either statement. For the PRINT# USING statement, the file number is the number you give the file when you open it.

For both statements, the string expression in quotation marks is a list of characters that determines the fields and formats of printed data. The list of expressions contains the items to print, separated by commas or semicolons. If the list ends with a semicolon, the cursor is left at the end of the last expression.

The characters in the format specification are replaced by the data in the print list, unless they are literal characters.

The following tables summarize the ST BASIC formatting characters:

STRING FIELD FORMATTING CHARACTERS

Character	Explanation
!	Tells the statement to print the first character of each specified string.
\chars\	chars plus 2 indicates the total number of characters to print from the specified string.
&	Specifies a variable length string field.

NUMERIC FIELD FORMATTING CHARACTERS

Character	Explanation
#	Represents each digit position in a numeric field.
.	Inserts a zero to fill digit positions as necessary.
+	Prints the sign of the number, plus or minus, before the printed number.
–	Prints negative numbers with a trailing minus sign.
**	Fills leading spaces in the numeric field with asterisks.
$$	Prints a dollar sign to the immediate left of the printed number.

**$	Fills leading spaces with asterisk and inserts a dollar sign to the left of the number.
,	Inserts a comma between every third digit on the left side of the decimal point.
^^^^	Specifies exponential format.
_	Prints the next character as a literal character.

You can include string constants in the format string, as shown in the following example.

Example:

```
Ok 10 PRINT USING "THIS IS FILE -###";4
Ok RUN
THIS IS FILE # 4
Ok
```

PUT

The PUT statement writes a record from a buffer to a random disk file.

Syntax:

PUT [#]<*file number*>,<*record number*>

```
PUT #1,5
```

Explanation:

The file number is the number you gave the file when you OPENed it. The record number is optional. If included, the record number must begin at one and proceed in sequential order. A FOR TO NEXT loop is an ideal way to assign record numbers in a file. If you do not give a record number, PUT uses the next record number in sequence after the last GET or PUT. The largest valid record number is 32767.

You should use LSET or RSET before a PUT to place the data into the random buffer.

Example:

```
Ok 100 LSET Q$=X$
Ok 120 PUT#2,RCORD%
```

QUIT

The QUIT command leaves ST BASIC and returns to GEM.

Syntax:
QUIT

```
QUIT
```

Explanation:
QUIT closes all files and returns you to GEM command level. Any program in memory is lost.

Same as: SYSTEM

Example:

```
Ok QUIT
```

RANDOMIZE

The RANDOMIZE statement seeds the random number generator.

Syntax:
RANDOMIZE [<*numeric expression*>]

```
RANDOMIZE X
```

Explanation:
You use RANDOMIZE with the RND function to generate random numbers. If you omit the optional numeric expression, ST BASIC asks for a random seed number on which to base RANDOMIZE.

If you do not use RANDOMIZE with a zero as a parameter at the beginning of a program that relies on random numbers, the RND function returns the same sequence of numbers every time you run the program.

See: RND function for further information on generating random numbers.

Example:

```
Ok 10 RANDOMIZE 0
Ok 20 FOR X=1 TO 10
Ok 30 PRINT RND
Ok 40 NEXT X
Ok RUN
 .957395
 .427143
 .806267
 .0206223
 .86628
 .886706
 .435054
 .199773
 .505868
 .801594
Ok
```

READ

The READ statement assigns values from a DATA statement to variables.

Syntax:

READ <*variable*>,<*variable*>

```
READ A,B,A$
```

Explanation:

The READ statement and DATA statement are always used together. READ assigns the values listed in DATA to a corresponding list of variables one by one. The variables can be numeric or string. They must agree in type with the constant values in the DATA statement; otherwise, an error results.

You can use one READ with several DATA statements, or vice versa. If the number of values in the DATA statement is greater than the number of variables in the READ statement, the next READ statement picks up the remaining constants where the first left off, and assigns them to the variables in its list. If there are no subsequent READ statements, the extra data is ignored.

If there are fewer values in the DATA statement than in the READ statement, the next data statement is found and read. If there is none, an out-of-data error results.

You can use the RESTORE statement to reread DATA items from the start of a specified line number.

See: DATA, RESTORE

Example:

```
Ok 10 READ X,Y,Z
Ok 20 RESTORE
Ok 30 AVERAGE=(X+Y+Z)/3
Ok 40 DATA 23.4,89.2,77
Ok 50 PRINT AVERAGE
Ok 60 READ X,Y,Z
Ok 70 PRODUCT=X*Y*Z
Ok 80 PRINT PRODUCT
Ok 90 END
Ok RUN
 63.2
 160720
Ok
```

REM

The REM statement introduces a remark.

Syntax:

REM <*remark*>

```
REM THIS IS A REMARK
' THIS IS A REMARK
```

Explanation:

REM statements help clarify the logic of a program. Remarks appear in the program listing as written, but they are not executable. Remarks can be as long as 245 characters. If you write a remark longer than the width of the screen, you can extend the line with a line feed.

If you branch into a REM line with a GOTO or GOSUB statement, the program continues executing at the first executable line after the REM.

The single apostrophe character has the same effect as REM. For example,

```
Ok 100 'this is a comment
```

is a valid statement.

Example:

```
Ok 10 REM THIS PROGRAM FINDS THE SQUARE OF A NUMBER
Ok 20 INPUT "ENTER A NUMBER TO BE SQUARED";X
Ok 30 S=X*X
Ok 40 PRINT S
Ok 50 'RETURN FOR ANOTHER NUMBER
Ok 60 GOTO 20
Ok 70 END
RUN
.
.
.
```

RENUM

The RENUM statement renumbers program lines.

Syntax:

RENUM [<*new first line*>][,<*starting line*>][,<*increment*>]

```
RENUM 50,10,20
```

Explanation:

If your program line numbers are irregular because you have inserted new lines between existing lines, you can renumber the entire program without having to change GOTO or other address-dependent statements.

Used alone, RENUM numbers the first line of the program 10, and increments succeeding lines by 10.

You can supply a new first line number. You can also supply a starting line, which is the current line number where you want the renumbering to begin.

You can also specify an increment for line numbering. For example:

```
RENUM 10,30,10
```

begins numbering at the old line 30, assigns it the line number 10, and sets an increment of 10. The following line numbers are 20, 30, 40, and so on.

You can also specify an increment for line numbering. For example:

```
RENUM 10,30,20
```

begins numbering at the old line 30, assigns it the line number 10, and sets an increment of 20. The following line numbers are 30, 50, 70, and so on.

You can use any of the RENUM options alone. However, if you specify only an increment, leave commas as place markers to show you are supplying an incremental value rather than a new first number or new first line. For example, RENUM ,,20.

RENUM adjusts all line number references in GOTO, GOSUB, IF . . . THEN . . . ELSE, ON . . . GOTO, and ON . . . GOSUB statements to reflect the new line numbers. If you have a nonexistent line in one of these statements, it remains unchanged.

You cannot use RENUM to change the order of program lines.

RENUM creates a file called BASIC.WRK on the current disk.

Note: The disk must not be write protected.

Example:

```
Ok 15 X=5
Ok 20 Z=3
Ok 25 Y=10
Ok 30 PRINT X+Y-Z
Ok RENUM
   LIST
10 X=5
20 Z=3
30 Y=10
40 PRINT X+Y-Z
Ok
```

REPLACE

The REPLACE statement replaces an old version of a file with a new version.

Syntax:

REPLACE [<*filename*>][,<*line number list*>]

```
REPLACE MYPROG.BAS
REPLACE MYPROG.BAS,100-800
```

Explanation:

You use REPLACE with OLD or LOAD. After you have loaded an old file and revised it, REPLACE sends the revised version onto disk, replacing the old version.

If you specify a filename, REPLACE saves the source program in <*filename*>, rather than the current program name. You can save parts of a program by specifying a line number list.

REPLACE works exactly like SAVE, except that with REPLACE, the name of the file you want to save can already belong to another file. The example brings program COUNTPROG into working storage, adds or replaces line 130, and stores the revised program in permanent storage on disk.

Example:

```
Ok OLD COUNTPROG
Ok 130 IF X=10 THEN END
Ok REPLACE
Ok
```

RESET

The RESET statement places the contents of the Output Window into the graphics buffer.

Syntax:

RESET

```
RESET
```

Explanation:

When buffered graphics is enabled, RESET duplicates the current contents of the Output Window in the graphics buffer. This allows a graphics image to be stored onto disk, or restored to the output window after subsequent graphics operations. The OPENW statement restores the contents of the graphics buffer to the Output Window.

Example:

```
10 COLOR 1,1,1,1,1:FULLW 2
20 CIRCLE 100,100,50
30 RESET: ' PUTS IMAGE INTO BUFFER
40 CLEARW 2
50 PCIRCLE 100,100,50
60 FOR I=1 TO 1000:NEXT
70 OPENW 2
80 END
```

RESTORE

The RESTORE statement rereads DATA statements.

Syntax:

RESTORE *<line descriptor>*

```
RESTORE 200
```

Explanation:

RESTORE lets you specify which DATA statement to use with READ statements. RESTORE finds the first item in the first DATA statement at or after the specified line and establishes it as the starting point for the next READ statement.

You can specify any DATA statement in a program as the object of a RESTORE statement by giving its line number. The line descriptor you give with RESTORE does not have to refer to DATA statement, or even exist. The next READ statement finds the next DATA statement after or equal to the line descriptor specified.

Example:

```
Ok 10 READ X,Y,Z
Ok 20 RESTORE
Ok 30 AVERAGE=(X+Y+Z)/3
Ok 40 DATA 23.4,89.2,77
Ok 50 PRINT AVERAGE
Ok 60 READ X,Y,Z
Ok 70 PRODUCT=X*Y*Z
Ok 80 PRINT PRODUCT
Ok 90 END
Ok RUN
 63.2
 160720
Ok
```

RESUME

The RESUME statement continues execution after an error.

Syntax:

RESUME (0)
RESUME NEXT
RESUME *<line descriptor>*

```
RESUME (0)
RESUME NEXT
RESUME 200
```

Explanation:

After an error has been detected and trapped, RESUME restores the program to normal execution. You write a RESUME statement at the end of an error trapping routine, and only there. A RESUME statement executed anywhere except in an active error trap causes an untrappable error.

RESUME used by itself or followed by a zero sends program control back to the statement where the error occurred.

RESUME NEXT sends program control to the statement following the one that caused the error.

RESUME <*line descriptor*> sends program control to a given line number.

Example:

```
Ok 100 ON ERROR GOTO 700
  .
  .
  .
Ok 700 IF (ERR=300) AND (ERR=150) THEN PRINT
"MINIMUM NUMBER OF DEPENDENTS IS 1":RESUME 140
```

RETURN

The RETURN statement transfers control from a subroutine to the statement following the last GOSUB.

Syntax:

RETURN

```
RETURN
```

Explanation:

RETURN transfers execution of a program to the first executable statement in the main program following a subroutine call. The subroutine call can be a GOSUB or ON . . . GOSUB statement.

Example:

```
Ok 10 GOSUB ALPHA
Ok 20 REM RETURN POINT OF SUBROUTINE
Ok 30 PRINT A
Ok 40 GOTO 200
Ok 100 ALPHA:REM START OF SUBROUTINE
Ok 110 A=5*6
Ok 120 RETURN
Ok 200 END
Ok RUN
 30
Ok
```

RIGHT$

The RIGHT$ function returns the rightmost characters of a string.

Syntax:

X$ = RIGHT$(<*target string*>, <*number of characters*>)

```
X$=RIGHT$(A$,5)
```

Explanation:

RIGHT$ assigns the number of characters you specify on the right of a target string to a new string variable. If the number of characters you ask for is greater than or equal to the length of the string, the entire string returns. If you ask for zero characters, a null string returns.

Example 1:

```
Ok 10 A$="Marketing Strategies"
Ok 20 B$="Regional Response"
Ok 30 C$="Test Results"
Ok 40 INPUT "CATALOG NUMBER";CATALOG$
Ok 50 IF RIGHT$(CATALOG$,1)="1" THEN PRINT "YOU HAVE
CHOSEN"
Ok 60 PRINT "TESTPRO CATALOG SERIES1"
Ok 70 PRINT "PLEASE CHOOSE FROM THE FOLLOWING
HEADINGS:"
Ok 80 PRINT A$
Ok 90 PRINT B$
Ok 100 PRINT C$
Ok RUN
CATALOG NUMBER? CASPAR BLEEBLEBOX CATALOG 201
YOU HAVE CHOSEN
TESTPRO CATALOG SERIES1.
PLEASE CHOOSE FROM THE FOLLOWING HEADINGS:
Marketing Strategies
Regional Response
Test Results
Ok
```

Example 2:

```
10 A$="ST BASIC"
20 B$=RIGHT$(A$,5) 30 PRINT B$
RUN
BASIC
Ok
```

RND

The RND function generates and returns a random number.

Syntax:

X = RND [(<*numeric expression*>)]

```
X=RND
X=RND(Y)
X=RND(0)
X=RND(-Y)
```

Explanation:

RND returns a uniformly distributed random number in the open interval between zero and 1. Unless you write a RANDOMIZE statement before the RND statement, the same sequence of random numbers generates on every run.

RND acts differently depending upon whether the numeric expression evaluates to a positive number, negative number, or zero.

RND (<*positive expression*>) returns the next number in the current sequence.

RND (0) returns the last random number generated, without affecting the current sequence.

RND (<*negative expression*>) reseeds the random number generator with the negative number and returns the first random number in the new sequence.

The numeric expression is optional. If you do not give one, RND acts as if you had given a positive expression as an argument.

Note: See RANDOMIZE for information about seed number.

Example:

```
Ok 10 RANDOMIZE
Ok 20 X=RND
Ok 30 ROLL$="TAILS"
Ok 40 IF X>.5 THEN ROLL$="HEADS"
Ok 50 INPUT "HEADS OR TAILS";R$
Ok 60 IF R$=ROLL$ THEN PRINT "YOU WIN" ELSE PRINT
"YOU LOSE"
Ok RUN
```

```
Random number seed (-32768 to +32767)? 2
HEADS OR TAILS? TAILS
YOU WIN
OK
```

RSET

The RSET statement moves a string into a specified string variable without reassigning the string variable.

Syntax:

RSET <*string variable*> = <*string expression*>

```
RSET A$=B$
```

Explanation:

RSET is commonly used to move data to file buffers by resetting them into variables dropped into file buffers by a previous FIELD statement.

If the string being moved is shorter than the destination, RSET right justifies the string and pads the left with spaces. If the string is longer than the destination, RSET ignores the extra characters.

You must RSET or LSET numbers before you can use them with MKS$, MKI$, or MKD$.

Example:

```
Ok 10 OPEN "R",#3,"TEST"
Ok 20 FIELD #3,20 AS A$,20 AS B$
Ok 30 RSET A$=X$
Ok 40 RSET B$=STRESS$
```

RUN

The RUN command begins program execution.

Syntax:

RUN
RUN <,*line descriptor*>
RUN <*filename*>

```
RUN
RUN ,200
RUN MYPROG.BAS
```

Explanation:

RUN executes a program currently in memory or in a disk file. Program execution begins with the first line of the program unless you specify otherwise. When the program to be run is in a disk file, RUN clears any current program from memory before loading the specified program.

Program output appears in the Output Window.

To stop program execution and enter the Break mode, type **[Control] [G]** or click the Break option in the Run menu.

To continue program execution, type CONT or press **[Return]**.

To exit the Break mode and discontinue program execution, type STOP or END. To discontinue program execution and return to ST BASIC, type **[Control] [C]**.

SAVE

The SAVE command saves program lines to disk.

Syntax:

SAVE [<*filename*>], [<*line descriptor list*>]

```
SAVE MYFILE
SAVE MYFILE, 20-30
SAVE MYFILE, 10,30,70,80
SAVE MYFILE, -30
```

Explanation:

SAVE puts a program, or specified lines from it, into a disk file. SAVE assumes file type .BAS unless you specify otherwise. If you attempt to SAVE a program using a name already on the disk, an error occurs. SAVE will not replace a disk file with a current program.

Use REPLACE to save a program into an existing disk file.

SGN

The SGN function returns the sign of a number.

Syntax:

X = SGN(<*numeric expression*>)

```
X=SGN(Y)
```

Explanation:

SGN returns 1 if the numeric expression is positive; -1 if the expression is negative; and 0 if the expression evaluates to zero.

Example:

```
Ok 10 X=SGN(-3)
Ok 20 Y=SGN(0)
Ok 30 Z=SGN(2)
Ok 40 PRINT X
Ok 50 PRINT Y
Ok 60 PRINT Z
Ok RUN
-1
 0
 1
Ok
```

SIN

The SIN function returns the sine of its argument expressed in radians.

Syntax:

X = SIN(<*numeric expression*>)

```
X=SIN(Y)
```

Explanation:

The SIN function assumes the expression is an angle in radians. To convert degrees to radians, multiply by pi/180, where pi = 3.141593. SIN converts integers to real numbers and returns a real number.

Example:

```
Ok 10 PRINT SIN(23)
Ok RUN
-.84622
Ok
```

SOUND

The SOUND statement controls the 3 sound channels.

Syntax:

SOUND <*numeric expression*>, <*numeric expression*>, <*numeric expression*>, <*numeric expression*>, <*numeric expression*>

```
SOUND VOICE, VOLUME, NOTE, OCTAVE, DURATION
```

Explanation:

SOUND makes musical notes.

VOICE is the number of the sound channel used (1-3).

VOLUME controls loudness (0 = OFF, 15 is loudest).

NOTE and OCTAVE control the pitch of a note. You select an octave number from 1 to 8 and a note number from 1 to 12. The note numbers correspond to the note positions in a musical scale. A 440 Hz A is note 10 in octave 4.

DURATION is the time in 1/50 second counts that a note will be held before the beginning of the next sound. The last sound statement for each voice should turn off the sound (e.g., SOUND 3,0,0,0,0). You can use the SOUND statement as a timing function by setting volume to 0 and the duration to the delay you want.

Example:

```
10 SOUND 1,8,12,4,25
20 SOUND 1,8,9,4,25
30 SOUND 1,0,0,0,0
```

SPACE$

The SPACE$ function returns a string of spaces.

Syntax:

X$ = SPACE$(<*numeric expression*>)

```
X$=SPACE$(Y)
```

Explanation:

SPACE$ returns as many spaces as you specify in the numeric expression. The value of the expression must be from 0 to 255.

Note: If you want to generate a number of spaces purely for printing, it is more efficient to use the SPC (X) function.

Example:

```
Ok 10 X=10
Ok 20 FOR V=1 TO 5
Ok 30 PRINT SPACE$(X);"1"
Ok 40 NEXT V
Ok 50 FOR Z=1 TO 21
Ok 60 PRINT "-";
Ok 70 NEXT Z
Ok RUN
          |
          |
          |
          |
          |
          |
----------
```

SPC

The SPC function outputs spaces to a PRINT statement.

Syntax:

PRINT SPC(<*numeric expression*>)

```
PRINT SPC(X)
```

Explanation:

SPC prints the number of spaces you specify in the numeric expression. The expression must evaluate to the range -32768 to +32767.

If the number of spaces you specify is greater than the declared width of the printer, the value used is the numeric expression MOD width.

For example, if the width is 72 and the numeric expression equals 100, SPC will insert 28 spaces.

If the numeric expression is greater than 255, the number of spaces inserted is the numeric expression MOD 255.

Note: Use SPC only with PRINT, LPRINT, and PRINT#.

Example:

```
Ok 10 PRINT "ALPHABET"
Ok 20 PRINT
Ok 30 PRINT "A"SPC(3)"a"SPC(7)
"B"SPC(3)"b"SPC(7)"C"SPC(3)"c"
Ok RUN
ALPHABET
A a   B b   C c
```

SQR

The SQR function returns the square root of a number.

Syntax:

X = SQR(<*numeric expression*>)

```
H=SQR(Y)
```

Explanation:

The number must not be negative. SQR returns a real number.

Example:

```
Ok 10 PRINT SQR(9)
Ok RUN
 3
Ok
```

STEP

The STEP command executes a program line by line.

Syntax:

STEP
STEP <,*line descriptor*>
STEP <*filename*>

```
STEP
STEP,200
STEP MYPROG.BAS
```

Explanation:

STEP runs a program one line at a time, printing each line along with any output and waiting for a **[Return]** before proceeding to the next line.

To exit from STEP, use the CONT command to begin normal execution, or the END command to stop altogether.

Example:

```
Ok 10 X=9
Ok 20 PRINT X
Ok 30 PRINT "HOW DO YOU DO?"
Ok 40 END
Ok STEP,10
S 10  X=9
BR [Return]
S 20  PRINT X
BR [Return]
S 30  PRINT "HOW DO YOU DO?"
BR [Return]
HOW DO YOU DO?
S 40 END
BR [Return]
OK
```

STOP

The STOP statement stops program execution and transfers the control of ST BASIC to the Command Window.

Syntax:

STOP

```
STOP
```

Explanation:

After a STOP, the program is at Break level. You can stop a program anywhere. Unlike END, STOP leaves files open, enters Break mode, and can be continued. It also prints the message "STOP".

CONT or **[Return]** resumes program execution.

Example:

```
Ok 10 A=4:B=6:C=8
Ok 20 PRINT A,A*B
Ok 30 STOP
Ok 40 PRINT C*A
Ok 50 END
Ok RUN
 4    24
Stop at line 30
Br CONT
 32
Ok
```

STR$

The STR$ function returns a string containing the decimal character representation of its argument.

Syntax:

X$ = STR$(<*numeric expression*>)

```
X$=STR$(Y)
```

Explanation:

The string returned contains the standard representation of the expression. It contains the characters that would print if a PRINT statement were executed.

For positive numbers, STR$ adds a leading blank for the plus sign, and STR$ deletes any space that follows a number.

VAL is the complementary function to STR$.

See: OCT$, HEX$

Example:

```
Ok 10 ZIPCODE=91899
Ok 20 PRINT STR$(ZIPCODE)+"(CALIFORNIA)"
Ok RUN
 91899 (CALIFORNIA)
Ok
```

STRING$

The STRING$ function returns a string of a given length. The characters are defined by the second argument.

Syntax:

X$ = STRING$(<*numeric expression*>, <*numeric or string expression*>)

```
X$=STRING$(Y,A$)
X$=STRING$(Y,N)
```

Explanation:

The first numeric expression is the length of the string that STRING$ returns. It must be in the range 0 to 255.

You can use a numeric or string expression for the second parameter. A numeric expression must be an ASCII code for a character. A string character can be of any kind.

STRING$ returns a string of the specified length consisting of the character for the specified ASCII code or the first character of the string expression.

STRING$ produces less memory fragmentation and works significantly faster than concatenation. When building a string containing a number of different characters, it is more efficient to use STRING$ or SPACE$ to create a string of the required length and then use MID$ to move individual characters into the string than to concatenate strings.

Example:

```
Ok 10 Z$=STRING$(20,"*")
Ok 20 PRINT Z$
Ok RUN
********************
Ok
```

SWAP

The SWAP statement trades the values of two variables.

Syntax:

SWAP <*first variable*>,<*second variable*>

```
SWAP X,Y
```

Explanation:

You can swap any type of variable, but the variables must be of the same type. You can swap array variables, but not arrays themselves:

`SWAP A%(3),B%(7,5)` is okay
`SWAP A%(),B%()` doesn't work

Example:

```
Ok 10 X$="TOM BRENTMEYER"
Ok 20 Y$="SUSAN STEIGER"
Ok 30 O$="FORMER"
Ok 40 C$="CURRENT"
Ok 50 M$="MARKETING MANAGER:"
Ok 60 PRINT O$;M$;X$
Ok 70 SWAP X$,Y$
Ok 80 SWAP O$,C$
Ok 90 PRINT O$;M$;X$
Ok RUN
FORMER MARKETING MANAGER: TOM BRENTMEYER
CURRENT MARKETING MANAGER: SUSAN STEIGER
Ok
```

SYSTAB

The SYSTAB is the beginning memory location of a table of system parameters and pointers.

Syntax:

X = PEEK(SYSTAB + OFFSET)

```
X=PEEK(SYSTAB+OFFSET)
```

Explanation:

With the exception of SYSTAB + 2, which is a READ/WRITE location, SYSTAB is a READ/ONLY location. Except for SYSTAB + 20,

the graphics buffer pointer, SYSTAB contains 2-byte values. SYSTAB + 20 contains a 4-byte long integer address.

The graphics buffer is 32768 bytes long. SYSTAB is organized as follows:

Offset	Function
0	Graphics Resolution (Planes) 1 = HI, 2 = MED, 4 = LO
2	Editor Ghost Line Style.
*4	EDIT AES Handle
*6	LIST AES Handle
*8	OUTPUT AES Handle
*10	COMMAND AES Handle
12	EDIT Open Flag (0 = CLOSED, 1 = OPEN)
14	LIST Open Flag (0 = CLOSED, 1 = OPEN)
16	OUTPUT Open Flag (0 = CLOSED, 1 = OPEN)
18	COMMAND Open Flag (0 = CLOSED, 1 = OPEN)
20	Graphics Buffer (4 byte pointer to 32768 byte buffer when BUFFERED GRAPHICS enabled)
**24	GEMFLAG (0 = NORMAL, 1 = OFF)

Bit	Description
0	Thickened
1	Intensity
2	Skewed
3	Underlined
4	Outline
5	Shadow

* Use of these handles requires knowledge of the TOS Operating System.

** GEMFLAG can be used to turn ST BASIC's interaction with GEM off to increase processing speed. When ST BASIC is off, no ST BASIC functions involving the screen, mouse, or keyboard will work. Disk I/O and processing functions are available. Your program must turn the interaction on again before it can take any form of user input.

Example:

```
POKE SYSTAB+24,1:'** TURNS OFF GEM
POKE SYSTAB+24,0:'** TURNS ON GEM
```

SYSTEM

The SYSTEM command leaves ST BASIC and returns to GEM.

Syntax:
SYSTEM

```
SYSTEM
```

Explanation:
SYSTEM closes all files and returns you to GEM command level. Any program in memory is lost.

Same as: QUIT

Example:

```
Ok SYSTEM
```

TAB

The TAB function moves the cursor to a specified tab position.

Syntax:
PRINT TAB(<*tab position*>)

```
PRINT TAB(Y)
```

Explanation:
TAB is used with PRINT, LPRINT, and PRINT#.

The tab position must evaluate to the range -32768 to + 32767. If the current print position is already beyond the tab position you specify, TAB goes to the next line and stops at the tab position you specify. The leftmost position is space 1; the rightmost is defined by a WIDTH statement. If the position evaluates to greater than 255, the position is computed Mod 256. If the position is greater than or equal to the width, it is computed Mod (width).

Example:

```
Ok 10 PRINT "1985 QUARTERLY EARNINGS"
Ok 20 PRINT
Ok 30 PRINT TAB(10)"WINTER"
Ok 40 PRINT TAB(70)"TOO FAR"
Ok 50 PRINT TAB(100)"SUMMER"
Ok 60 END
Ok RUN
1985 QUARTERLY EARNINGS
   WINTER
TOO FAR
      SUMMER
```

TAN

The TAN function returns the tangent of a number.

Syntax:

X = TAN(<*angle in radians*>)

```
X=TAN(Y)
```

All ST BASIC trigonometric functions require that you specify angles in radians.

Explanation:

The TAN function operates on radian values and returns a real number. To convert degrees to radians, multiply them by pi/180, where pi = 3.141593.

Example:

```
Ok 10 RADIAN!=34
Ok 20 TANGENT!=TAN(RADIAN!)
Ok 0 PRINT TANGENT!
RUN
-.6235
Ok
```

TRACE

The TRACE command follows program execution line by line and selectively prints the entire line.

Syntax:

TRACE [<*line descriptor list*>]

```
TRACE
TRACE 20,40
TRACE 20-40
TRACE -40
```

Explanation:

You can use the TRACE command during debugging to print program lines as they run.

TRACE prints each line before executing it.

TRACE 20, 40 prints lines 20 and 40 each time they execute.

TRACE 20-40 prints lines 20 through 40 each time they execute.

UNTRACE cancels TRACE.

See: TRON, FOLLOW

Example:

```
Ok 10 FOR X=1 TO 2
Ok 20 N=N+1
Ok 30 B=B+1
Ok 40 PRINT N
Ok 50 PRINT B
Ok 60 NEXT X
Ok RUN
 1
 1
 2
 2
Ok TRACE
Ok RUN
T 10 FOR X=1 TO 2
T 20 N=N+1
T 30 B=B+1
T 40 PRINT N
 1
```

```
T 50 PRINT B
 1
T 60 NEXT X
T 20 N=N+1
T 30 B=B+1
T 40 PRINT N
 2
T 50 PRINT B
 2
T 60 NEXT X
Ok UNTRACE
Ok
```

TROFF

The TROFF command cancels the TRON command.

Syntax:
TROFF [<*line descriptor list*>]

```
TROFF
TROFF 10,40
TROFF 10-40
TROFF -40
```

Explanation:
TROFF cancels TRON either completely or for selected lines.

See: TRON

TRON

The TRON command selectively traces program execution line by line and prints the line numbers.

Syntax:
TRON [<*line descriptor list*>]

```
TRON
TRON 20,40
TRON 20-40
TRON -40
```

Explanation:

Use TRON during debugging to follow the course of the program line by line.

TRON prints each line number of the program as it executes and traces the values of variables. The line descriptor appears in square brackets.

TROFF cancels TRON.

See: TRACE, FOLLOW

Example:

```
Ok 10 FOR X=1 TO 3
Ok 20 N=N+1
Ok 30 B=B+1
Ok 40 PRINT N
Ok 50 PRINT B
Ok 60 NEXT X
Ok RUN
 1
 1
 2
 2
 3
 3
Ok TRON
Ok RUN
[10]
[20]
[30]
[40]  1  (Appears in Output Window)
[50]  1  (Appears in Output Window)
[60]
[20]
[30]
[40]  2  (Appears in Output Window)
[50]  2  (Appears in Output Window)
[60]
[20]
[30]
```

```
[40]  3  (Appears in Output Window)
[50]  3  (Appears in Output Window)
[60]
Ok TROFF
Ok
```

UNBREAK

The UNBREAK command selectively cancels a BREAK command.

Syntax:
UNBREAK [<*line, descriptor list*>]

```
UNBREAK
UNBREAK 20,50
UNBREAK -50
UNBREAK 20-50
```

Explanation:
UNBREAK cancels BREAK either completely or for selected lines.

See: BREAK

UNFOLLOW

The UNFOLLOW command cancels the FOLLOW command.

Syntax:
UNFOLLOW [<*variable*>],[<*variable*>]

```
UNFOLLOW
UNFOLLOW X,Y
```

Explanation:
UNFOLLOW cancels FOLLOW either completely or for selected variables.

See: FOLLOW

UNTRACE

The UNTRACE command cancels the TRACE command.

Syntax:

UNTRACE [<*line descriptor list*>]

```
UNTRACE
UNTRACE 10,40,70
UNTRACE 10-40
UNTRACE -40
```

Explanation:

UNTRACE cancels TRACE either completely or for selected lines.

See: TRACE

VAL

The VAL function scans a string of characters and converts them to a real number.

Syntax:

X = VAL(<*digit string expression*>)

```
X=VAL(A$)
```

Explanation:

VAL scans the string from left to right, skipping leading tabs, spaces, and line feeds, until it reaches the end of the string or finds a character that is not a digit. VAL scans strings in the same way that the INPUT# statement reads into a numeric variable.

If the first character of the string is not a valid part of a number, VAL returns a zero.

VAL is the complement to STR$.

Example:

```
Ok 10 READ ID$
Ok 20 IF VAL(ID$)<300 THEN 30
Ok 30 EXPIRATION$="JAN 1, 1985"
Ok 40 IF VAL(ID$)>300 THEN 50
Ok 50 EXPIRATION$="JAN 1, 1990"
```

VARPTR

The VARPTR function returns the address of a variable.

Syntax:

X = VARPTR(<*variable*>)
X = VARPTR(# <*file number*>)

```
X=VARPTR(Y)
X=VARPTR(#1)
```

Explanation:

You can use VARPTR to find the address of a variable so that you can pass it to an assembly language subroutine. The variable can be of any type, including array, but you must have assigned it a value before you can find its address with VARPTR. VARPTR returns a value which is the absolute address of the first byte of the named variable.

In the case of files, the file number is the number you assigned a disk file when you opened it. VARPTR returns the starting address of the file's input/output buffer.

Example:

```
Ok 50 X=VARPTR(MATERIALS)
```

VDISYS

The VDISYS function allows the user to access the operating system's VDI interface.

Syntax:

VDISYS(<*Dummy Argument*>)

```
VDISYS(1)
```

Explanation:

To access the VDI interface, POKE the CONTRL, INTIN, and PTSIN arrays with the proper values before making the VDISYS call. Output from the VDI level can be accessed through the INTOUT and PTSOUT arrays.

Example:

```
10 REM DRAW A CIRCLE AT 50,50 WITH RADIUS 25
20 COLOR 1,1,1,1,1 :FULLW 2
30 POKE CONTRL,11
40 POKE CONTRL+2,3
50 POKE CONTRL+6,0
60 POKE CONTRL+10,4
70 POKE PTSIN,50
80 POKE PTSIN+2,50
90 POKE PTSIN+8,25
100 VDISYS(1)
```

WAIT

The WAIT statement halts the program while waiting for an I/O port to develop a bit pattern.

Syntax:

WAIT <*port number*>,<*integer expression*>[,<*integer expression*>]

```
WAIT 200,H,Y
```

Explanation:

WAIT stops program execution until a given bit pattern develops in a machine input port. The logical operator XOR tests the data from the port to determine whether it corresponds to the optional second integer expression. If you omit the optional expression, it is assumed zero.

The AND operator then tests the data against the first integer expression. If the result of the text is zero, execution loops back and grabs the next data at the port. When the result is non-zero, execution goes on to the next statement.

If WAIT does not finds a bit pattern that results in zero, it loops infinitely, and you must reboot the machine.

Example:

```
Ok 100 WAIT 5,&H2,&H3
Ok 110 PRINT "NUMBER FOUND"
```

WAVE

The WAVE statement controls the waveforms used in SOUND statements.

Syntax:

WAVE <*numerical expression*>, <*numerical expression*>, <*numerical expression*>, <*numerical expression*>, <*numerical expression*>,

```
WAVE ENABLE, ENVELOPE, SHAPE, PERIOD, DELAY
```

Explanation:

ENABLE is the mixer register of the sound generator. A 0 on bits 0-2 enables voice 1-3. A 0 on bits 3-5 places the noise on voice 1-3. More than one voice can be selected at once.

ENVELOPE is the envelope generator register. A 1 on bits 0-2 enables the envelope for voice 1-3. More than one can be enabled.

SHAPE is the envelope shape and cycle control register. Bits 0-3 are used as shown in the chart on the next page.

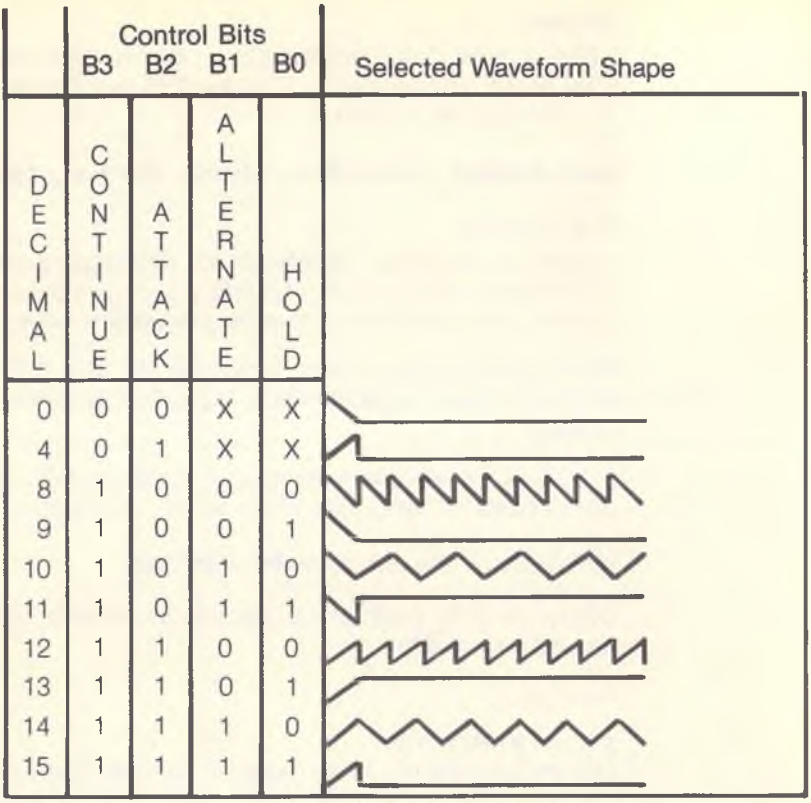
PERIOD sets the period of the envelope.

DELAY sets the time in 1/50 second increments before BASIC resumes execution.

Example:

```
5 REM THANKS R.K.
10 FOR I=1000 to 1200:WAVE I,I/100,1000,1000,10:NEXT
```

REGISTER $OD WAVEFORM CONTROL



0 - Off
1 - On
X - Not Used

WEND

The WEND statement signals the end of a WHILE/WEND loop.

Syntax:

WEND

```
WEND
```

Explanation:

WEND is used solely with WHILE to direct program flow back to the WHILE statement. A nested WEND associates with the nearest WHILE.

See: WHILE

Example:

```
Ok 10 X=8
Ok 20 WHILE X
Ok 30 PRINT "$";
Ok 40 X=X-1
Ok 50 WEND
Ok 60 END
Ok RUN
$$$$$$$$
Ok
```

WHILE

The WHILE statement states a condition that controls a WHILE/WEND loop.

Syntax:

WHILE *<logical expression>*

```
WHILE A<B
```

Explanation:

WHILE initiates a WHILE/WEND loop that continues running until the logical expression is false (i.e., 0). The statements between WHILE and WEND execute while the conditional expression in the WHILE statement is true.

The WEND statement at the end of the loop sends program flow back to the WHILE condition. The condition at the WHILE loop is evaluated and the loop repeats while the condition is true (not zero). When the condition is false, execution continues at the statement following WEND.

You can nest WHILE/WEND loops. Each WEND matches the most recent WHILE. A WHILE without a WEND or a WEND without a WHILE causes an error.

See: WEND

Example:

```
Ok 10 M=10
Ok 20 P=5
Ok 30 WHILE M>P
Ok 40 PRINT "COUNT LOOP"
Ok 50 M=M-1
Ok 60 WEND
Ok 70 END
Ok RUN
COUNT LOOP
COUNT LOOP
COUNT LOOP
COUNT LOOP
COUNT LOOP
Ok
```

WIDTH

The WIDTH statement sets the line width of the screen or printer.

Syntax:

WIDTH [LPRINT] <*integer expression*>

```
WIDTH 72
WIDTH LPRINT 72
```

Explanation:

The default width of the screen and printer is 72 characters. You can change it with WIDTH.

The integer expression is the line width in characters; it must be in the range 14 to 255. The LPRINT option sets the line width for the printer. Otherwise, the line width is set for the screen.

When printing, ST BASIC prints a carriage return before any character that would otherwise print past the line width limit. To prevent unwanted carriage returns in your output, set the line width to 255. ST BASIC then assumes the device has infinite width and does not insert carriage returns.

See: POS, LPOS

Example;

```
Ok 10 WIDTH 33
Ok 20 FOR I=1 TO 50
Ok 30 PRINT "-";
Ok 40 NEXT
Ok RUN
-----------------
---------
Ok
```

WRITE

The WRITE statement outputs data to the terminal.

Syntax:

WRITE [<*expression*>],<*expression*>

```
WRITE X,Y,A$
```

Explanation:

Like PRINT, WRITE sends output to the screen, but WRITE prints commas between the items and quotation marks around strings.

Each item is separated from the next on the terminal with a comma.

String values print with quotation marks, and after the last item, the cursor spaces down to the start of the next line.

WRITE sends a blank line to the terminal if you do not specify a list of expressions to output.

See: PRINT, PRINT#

Example 1:

```
Ok 100 X$="HAPPY MOTORING"
Ok 110 Z=010583
Ok 120 WRITE Z
Ok 130 WRITE
Ok 140 WRITE X$
Ok RUN
 10583

"HAPPY MOTORING"
Ok
```

WRITE#

The WRITE# statement outputs data to a sequential file.

Syntax:

WRITE# [<*expression*>],<*expression*>

```
WRITE #1,X,Y,A$
```

Explanation:

WRITE# is similar to WRITE but sends the data to a sequential file, not the terminal. The file number is the number you opened the file with. You must have opened the file in O mode.

WRITE# is preferable to PRINT# when you plan to read the data back with a series of INPUT# statements. The output from WRITE# is in the form required to read back the data accurately.

The rules for forming the expression are the same as those for PRINT#.

See: PRINT, PRINT#

Example:

```
Ok 10 KWH=34.275
Ok 20 K$="AVERAGE KILOWATT HOURS PER WEEK"
OK 30 WRITE #2,K$,KWH
```

This writes to disk as:

```
"AVERAGE KILOWATT HOURS PER WEEK",34.275
```

Close the file, reopen for input, then read the file:

```
Ok 40 INPUT#2,K$,KWH

"AVERAGE KILWATT HOURS PER WEEK" to K$ and 34.275 to B$
```

APPENDIX D
ERROR MESSAGES

Number	Message
2	Something is wrong.
3	RETURN statement needs matching GOSUB.
4	READ statement ran out of data.
5	Function call not allowed.
6	Number too large.
7	Not enough memory.
8	A statement or a command refers to a nonexistent line.
9	Subscript refers to element outside the array.
10	You defined an array more than once.
11	You cannot divide by zero.
12	Statement is illegal in direct mode.
13	Types of values do not match.
14	Undefined error.
15	Strings cannot be over 255 characters long.
16	Expression is too long or too complex.
17	CONT works only in Break mode.
18	Function needs prior definition with DEF FN.
19	Undefined error.
20	RESUME statement found before error routine entered.
21	Undefined error.
22	Expression has operator with no following operand.
23	Program line too long.
24-29	Undefined error.
30	Window number invalid.
31	Argument out of range.
32	Command cannot be executed from the editor.
33	Line is too complex.
34-49	Undefined error.
50	FIELD statement caused overflow.
51	Device number invalid.
52	File number or filename invalid.
53	File not found on disk drive specified.
54	File mode is not valid.
55	You cannot OPEN or KILL a file already open.
56	Undefined error.
57	Disk input/output error.
58	File exists.

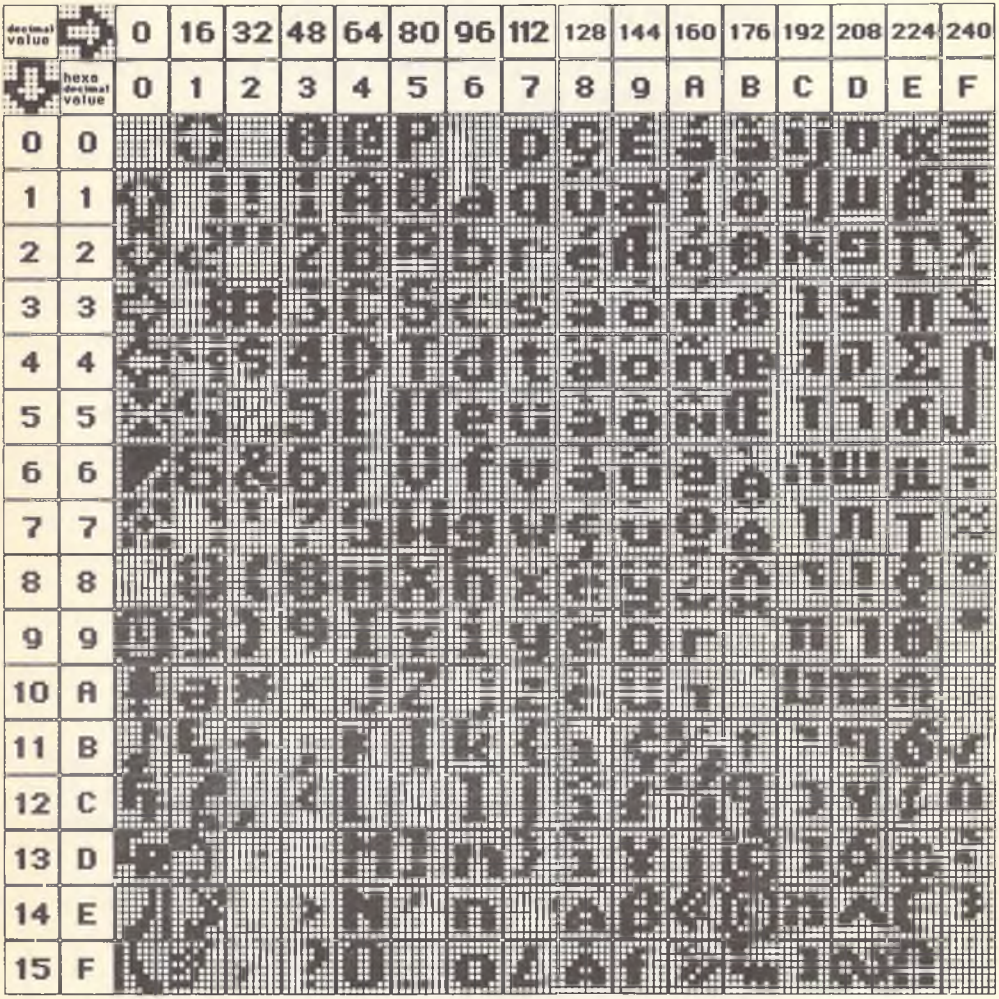
59-60	Undefined error.
61	Disk is full.
62	You have reached end-of-file.
63	The record number in PUT or GET is more than 32767 or zero.
64	Invalid filename.
65	Invalid character *character* in program file.
66	Program file has statement with no line number.
67-98	Undefined error.
99	—Break—.
100	Undefined error.
101	Program has too many lines.
102	ON statement is out of range.
103	Invalid line number.
104	A variable is required.
105	Undefined error.
106	Line number does not exist.
107	Number too large for an integer.
108	Input data is not valid, restart input from first item.
109	Stop.
110	You have nested subroutine calls too deep.
111	Invalid BLOAD file.
112-201	Undefined error.
202	Command not allowed here.
203	Line number is required.
204	FOR statement needs a NEXT or WHILE statement needs a WEND.
205	NEXT statement needs a FOR or WEND statement needs a WHILE.
206	A comma is expected.
207	A parenthesis is expected.
208	Option Base must be 0 or 1.
209	Statement end is expected.
210	Too many arguments in your list.
211	Not used.
212	Cannot redefine variable(s).
213	Function defined more than once.
214	You are trying to jump into a loop.
215-220	Undefined error.
221	System error #X, please restart.
222	Program not run.
223	Too many FOR loops.

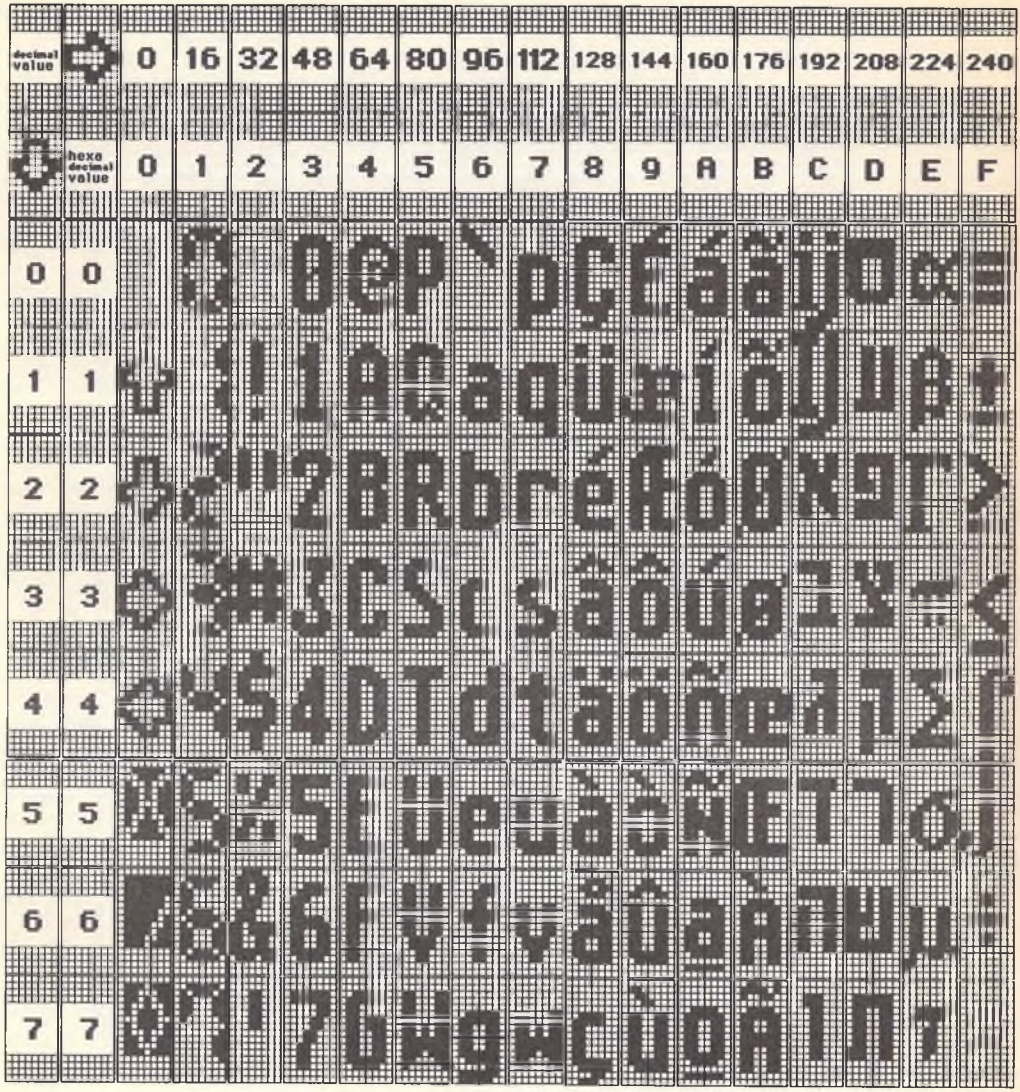
APPENDIX E
ST ASCII CHARACTER SET

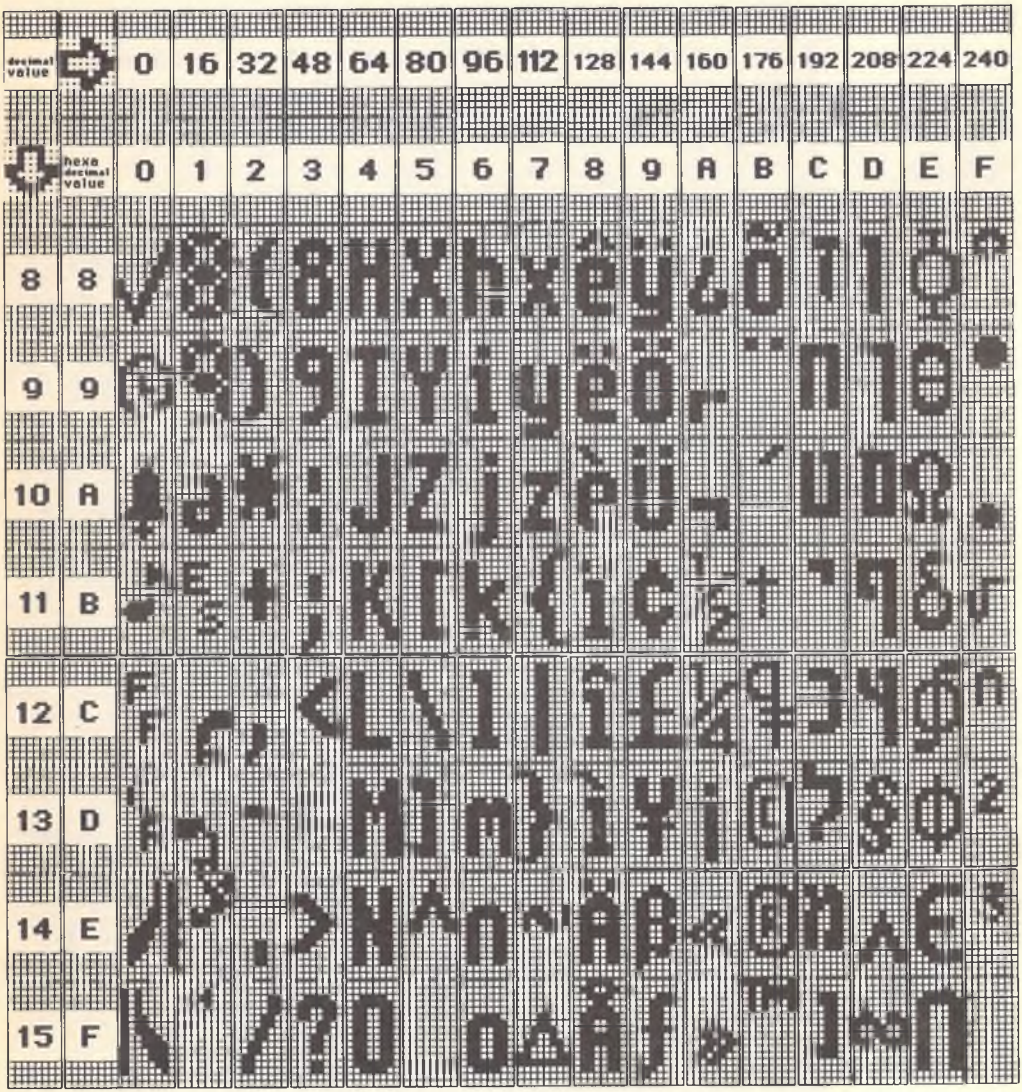
The following tables show the complete character sets available on the ST Computer. To print any of these characters from ST BASIC, input and run the following program:

```
  5  ' THIS PROGRAM PRINTS A LIST OF ALL ST ASCII
     CHARACTERS
  6  ' AND THEIR CODES.
 10  FULLW 2:CLEARM 2
 20  GOTOXY 1,2:?"LIST OF ST ASCII
     CHARACTERS":GOTOXY 0,4
 30  P=0:I=0
 40  FOR C=1 TO 255
 50  IF P>4 THEN P=0:I=I+1:?
 60  IF I=10 THEN GOTOXY 1,14: INPUT "PRESS
     [Return] TO CONTINUE...",A$
 70  IF I=10 THEN I=0:GOTOXY 0,4
 80  IF C=10 THEN ? "10=[Return]";:GOTO 120
 90  IF C=7 THEN ? "7=[Bell]";:GOTO 120
100  IF C=251 THEN GOTOXY 0,14
110  ? C;"= ";CHR$(C);" ";
120  P=P+1
130  NEXT C
140  GOTOXY 1,16: INPUT "PRESS [Return] TO
     EXIT...",A$
150  END
```

There are two character tables. The first is set up for 8 × 8 characters; the second for 8x16 characters. The different character set sizes are used with different screen resolutions.


decimal value
hexa decimal value
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 A
11 B
12 C
13 D
14 E
15 F


decimal value
hexa decimal value
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7


decimal value
hexa decimal value
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240
0 1 2 3 4 5 6 7 8 9 A B C D E F
8 8
9 9
10 A
11 B
12 C
13 D
14 E
15 F

APPENDIX F
ASSEMBLY LANGUAGE MODULES

The CALL statement in ST BASIC allows the use of assembly language modules. To use a module, you must load it into memory with a BLOAD statement, assign its starting address to a variable, and CALL it from ST BASIC (passing any necessary parameters to it).

Parameters are passed from ST BASIC to assembler programs in the following manner. The machine language module will find two parameters on the user stack (A7). The first is a 2-byte integer specifying the number of parameters being passed. (In the example below, it is 3.) The second is a 4-byte pointer to an array that contains the parameters. Each parameter in the array will occupy 8 bytes, regardless of its type. In the case of a string variable, the 8-byte value is a pointer to the string.

Before returning to ST BASIC, the assembler program can put any parameters it wants to pass to ST BASIC into a given memory location. Later, the ST BASIC program can PEEK at these parameters.

Example:

```
500  DIM A$(8):I%=70:X=22
510  CHART=18566:   'START ADDRESS OF THE
     ASSEMBLER LANGUAGE CODE
530  CALL CHART(I%, A$, X)
```

APPENDIX G
DERIVED FUNCTIONS

Derived Functions	Derived Functions in Terms of ATARI Functions
Secant	DEF FNSEC(X) = 1/COS(X)
Cosecant	DEF FNCSC(X) = 1/SIN(X)
Inverse Sine	DEF FNARCSIN(X) = ATN(X/SQR(-X*X + 1))
Inverse Cosine	DEF FNARCCOS(X) = -ATN(X/SQR(-X*X + 1) + CONSTANT
Inverse Secant	DEF FNARSEC(X) = ATN(SQR(X*X-1)) + (SGN(X-1)*CONSTANT
Inverse Cosecant	DEF FNARCCSC(X) = ATN(1/SQR(X*X-1)) + (SGN(X-1)*CONSTANT
Inverse Cotangent	DEF FNARCCOT(X) = ATN(X) + CONSTANT
Hyperbolic Sine	DEF FNSINH(X) = (EXP(X)-EXP(X))/2
Hyperbolic Cosine	DEF FNCOSH(X) = (EXP(X) + EXP(-X)/2
Hyperbolic Tangent	DEF FNTANH(X) = -EXP(-X)/(EXP(X) + EXP(-X))*2 + 1
Hyperbolic Secant	DEF FNSECH(X) = 2/(EXP(X) + EXP(-X))
Hyperbolic Cosecant	DEF FNCSCH(X) = 2/(EXP(X)-EXP(-X))
Hyperbolic Cotangent	DEF FNCOTH(X) = EXP(-X)/(EXP(X)-EXP(-X))*2 + 1
Inverse Hyperbolic Sine	DEF FNARCSINH(X) = LOG(X + SQR(X*X + 1))
Inverse Hyperbolic Cosine	DEF FNARCCOSH(X) + LOG(X + SQR(X*X-1))
Inverse Hyperbolic Tangent	DEF FNARCTANH(X) = LOG((1 + X)/(1-X))/2

Inverse Hyperbolic Secant	DEF FNARCSECH(X) = LOG ((SQR(-X*X + 1) + 1)/X)
Inverse Hyperbolic Cosecant	DEF FNARCCSCH(X) = LOG ((SGN(X)*SQR(X*X + 1) + 1)/X)
Inverse Hyperbolic Contangent	DEF FNARCCOTH(X) = LOG((X + 1)/(X-1))/2

Note: In this chart, the variable X in parentheses represents the value or expression to be evaluated by the derived function. Any variable name is permissible as long as it represents the number or expression to be evaluated.

APPENDIX H
SAMPLE PROGRAMS

BOXES

An interesting example of the RND statement using color graphics. Run this program in low-resolution color mode.

```
 10  ' FILL BOXES SYMMETRICALLY
 20  randomize 0:c=0
 30  color 1,0,1,1,1:fullw 2:clearw 2
 40  for x=18 to 284 step 19
 50  linef x,0,x,166
 60  next x
 70  for y=13 to 153 step 14
 80  linef 0,y,303,y
 90  next y
100  c=c+1:if c=16 then c=1
110  color 1,c,1
120  col=int(rnd*16)*19+9:row=int(rnd*12)*14+7
130  fill col,row,1
140  if col>151 then cenc=col-151:fill
     col-(cenc*2),row,1
150  if col<152 then colh=302-col:fill colh,row,1
160  if row>82 then rowh=row-((row-82)*2):fill
     col,rowh,1
170  if row<83 then rowh=164-row:fill col,rowh,1
180  if col>151 then fill col-(cenc*2),rowh,1 else
     fill colh,rowh,1
190  goto 100
```

CIRCLE OF PATTERNS

This program divides a circle into segments and then fills the segments with patterns. To vary the program, change line 120 to:

```
120  pellipse x,y,x,y,b,b+100
```

```
 10  ' CIRCLE WITH 36 PATTERNED SEGMENTS
 20  color 1,0,1,1,1:fullw 2:clearw 2
 30  if peek(systab)=1 then 60
 40  if peek(systab)=2 then 70
 50  goto 80
 60  x=386:y=172:s=170:goto 90
 70  x=304:y=83:s=182:goto 90
 80  x=151:y=83:s=91
 90  a=24:i=2:b=0
100  for p=1 to a
110  color 1,1,1,p,i
120  pcircle x,y,s,b,b+100
130  b=b+100
140  next p
150  if i=3 then end
160  i=3:a=12:goto 100
```

GRID OF PATTERNS

This program selects the screen resolution automatically, then displays 36 fill patterns.

```
 10  ' DISPLAY GRID WITH 36 FILL PATTERNS
 20  color 1,0,1,1,1:fullw 2:clearw 2
 30  if peek(systab)=1 then 60
 40  if peek(systab)=2 then 70
 50  goto 80
 60  x=102:y=56:a=28:b=308:c=56:d=51:e=561:
     f=102:goto 90
 70  x=102:w=28:a=14:b=154:c=28:d=51:e=561:
     f=102:goto 90
 80  x=51:y=28:a=14:b=154:c=28:d=25:e=280:f=51
 90  for x=f to e-d step f
100  linef x,0,x,345
110  next x
120  for y=c to b-a step c
130  linef 0,y,615,y
140  next y
150  i=2:p=1
 10  for y=a to b step c
170  for x=d to e step f
180  color 1,1,1,p,i:fill x,y,1
190  p=p+1:if p=25 then p=1:i=i+1
200  if i=4 then end
210  next x,y
```

LOW RESOLUTION DEMO

An interesting demonstration of low-resolution shapes and colors.

```
 10  color 1,0,1,1,1:fullw 2:clearw 2
 20  PIE: c=1
 30  for b=0 to 3360 step 240
 40  color 1,c,1
 50  pcircle 151,83,91,b,b+240
 60  c=c+1
 70  next b
 80  gosub DELAY
 90  OVAL: c=1
100  for b=0 to 3360 step 240
110  color 1,c,1
120  pellipse 151,83,151,83,b,b+240
130  c=c+1
140  next b
150  gosub DELAY
160  FILLPTNS: c=1:a=24:i=2
170  for p=1 to a
180  clearw 2
190  for x=61 to 244 step 61
200  linef x,0,x,166
210  next x
220  for y=55 to 110 step 55
230  linef 0,y,303,y
240  next y
250  y=2
260  for x=30 to 270 step 60
270  color 1,c,1,p,i
280  fill x,y,1
290  c=c+1:if c=16 then c=1
300  next x
310  y=y+55:if y=167 then 330
320  goto 260
330  next p
340  if i=3 then 360
350  a=12:i=3:goto 170
360  gosub DELAY
370  COLORFULCIRCLE: c=1:r=91
380  for b=0 to 3600 step 200
390  color 1,c,1
400  pcircle 151,83,r,b,b+200
```

```
410  c=c+1:if c=16 then c=1
420  next b
430  r=r-1:if r=0 then 450
440  goto 380
450  gosub DELAY
460  COLORFULELLIPSE:c=1:x=151:y=83
470  for b=0 to 3600 step 240
480  color 1,c,1
490  pellipse 151,83,x,y,b,b+240
500  c=c+1:if c=16 then c=1
510  next b
520  x=x-2:y=y-2:if y=3 then 540
530  goto 470
540  gosub DELAY
550  end
560  DELAY: for z=1 to 3000:next
570  color 1,0,1,1,1:clearw 2
500  return
```

MEDIUM RESOLUTION DEMO

This program demonstrates the medium-resolution color palette of your ST Computer.

```
 10  color 1,0,1,1,1:fullw 2:clearw 2
 20  PIE: c=1
 30  for b=0 to 3360 step 240
 40  color 1,c,1
 50  pcircle 304,83,182,b,b+240
 60  c=c+1:if c=4 then c=1
 70  next b
 80  gosub DELAY
 90  OVAL: c=1
100  for b=0 to 3360 step 240
110  color 1,c,1
120  pellipse 304,83,304,83,b,b+240
130  c=c+1:if c=4 then c=1
140  next b
150  gosub DELAY
160  FILLPTNS: c=1:a=24:i=2
170  for p=1 to a
180  clearw 2
190  for x=203 to 609 step 203
200  color 1,c,1,p,i
210  linef x,0,x,170
220  fill x-2,2
230  c=c+1:if c=4 then c=1
240  next x,p
250  if i=3 then 270
260  a=12:i=3:goto 170
270  gosub DELAY
280  end
290  DELAY: for z=1 to 3000:next
300  color 1,0,1,1,1:clearw 2
310  return
```

HIGH RESOLUTION DEMO

Show off your high-resolution monochrome monitor with this program!

```
 10  fullw 2:clearw 2
 20  SQUARES: a=2:b=3:L=61:w=56
 30  x=a:y=b
```

```
 40  linef x,y,x+L,y
 50  linef x+L,y,x+L,y+w
 60  linef x+L,y+w,x,y+w
 70  linef x,y+w,x,y
 80  x=x+61
 90  if x>600 then x=a:y=y+56
100  if y>320 then 120
110  goto 40
120  a=a+2:b=b+2:L=L-4:w=w-4
130  if w<0 then 150
140  goto 30
150  gosub DELAY
160  LINES: x=0:y=0
170  while x<614
180  linef 307,172,x,y
190  x=x+5
200  wend
210  while y<344
220  linef 307,172,x,y
230  y=y+3
240  wend
250  while x>0
260  linef 307,172,x,y
270  x=x-5
280  wend
290  while y>0
300  linef 307,172,x,y
310  y=y-3
320  wend
330  gosub DELAY
340  DESIGN: x1=1:x2=614:y1=1:y2=343
350  linef x1,y1,x2,y1
360  linef x2,y1,x2,y2
370  linef x2,y2,x1,y2
380  linef x1,y2,x1,y1
390  x1=x1+2:x2=x2-2:y1=y1+2:y2=y2-2
400  if y2>-22 then 350
410  gosub DELAY
420  end
430  DELAY: for z=1 to 5000:next
440  clearw 2:return
```

TRIGONOMETRY

Use this program to graph any trigonometric function.

```
 10  ' TRIG GRAPHS
 20  ' BY ROB COLLIER
 30  pi=3.1415926
 40  fullw 2:color 1,0,1:clearw 2
 50  SCREEN:
 60  if peek(systab)=4 then goto LOW
 70  if peek(systab)=2 then goto MEDIUM
 80  if peek(systab)=1 then goto HIGH
 90  INIT: t=0:L=0
100  lng=r/4:inc=pi/lng:off=b/4
110  FUNCTION: value=-2*pi
120  clearw 2
130  print "choose a function:":print
140  print "1) sine"
150  print "2) cosine"
160  print "3) tangent"
170  print "4) cosecant"
180  print "5) secant"
190  print "6) cotangent"
200  print:input choice
210  if choice>0 and choice<7 then goto GRAPH
220  ?"pick one of these numbers, please."
230  goto FUNCTION
240  PLOT:
250  value=-2*pi
260  x=L:x1=L:y1=b/2
270  on choice gosub
     SINE,COSINE,TANGENT,COSECANT,SECANT,COTANGENT
280  y=off*y:y=b/2-y
290  if y<t or y>b goto SKIP
300  if x<l or x>r goto SKIP
310  linef x1,y1,x,y
320  SKIP: x1=x
330  y1=y:x=x+1
340  value=value+inc
350  if value>2*pi then goto DONE
360  goto 270
370  DONE: input wait$
380  goto 120
```

```
390  GRAPH: color 1,bg,gr:clearw 2
400  linef l,b/2,r,b/2
410  linef r/2,t,r/2,b
420  color 1,bg,ln
430  goto PLOT
440  SINE: y=sin(value):return
450  COSINE: y=cos(value):return
460  TANGENT: y=tan(value):return
470  COSECANT: hold=sin(value)
480  if hold=0 then return
490  y=1/hold:return
500  SECANT: hold=cos(value)
510  if hold=0 then return
520  y=1/hold:return
530  COTANGENT:hold=tan(value)
540  if hold=0 then return
550  y=1/hold:return
560  LOW: r=303:b=167
570  gr=2:ln=14:bg=4
580  goto INIT
590  MEDIUM: r=608:b=167
600  gr=1:ln=2:bg=3
610  goto INIT
620  HIGH: r=615:b=343
630  gr=1:ln=1:bg=0
640  goto INIT
```

EFFECTIVE INTEREST RATE

Use this program to analyze finance packages.

```
 10  'Effective interest rate program by Richard
     Lauck
 20  'The program uses a form of Newton's method for
     estimating roots.
 30  'In effect the program uses calculas within
     the epsilon, "E" defined at line 100.
 40  'The formulas consider each payment to be made
     at the end of a period.
 50  clearw 2:fullw 2:?
 60  ? "FINAL LUMP SUM PAYMENT = ";:INPUT R
 70  ? "MONTHLY PAYMENT = ";:INPUT A
 80  ? "COST IF BOUGHT NOW = ";:INPUT C
 90  ? "NUMBER OF PAYMENTS = ";:INPUT N
100  Z=12:I=0.01:E=0.01:K=0
110  ?:PRINT " THE EFFECTIVE INTEREST RATE WITH: "
120  PRINT " "
130  PRINT "A FINAL LUMP SUM PAYMENT OF $";R
140  PRINT "A MONTHLY PAYMENT OF $";A
150  PRINT "AND PAYMENTS NUMBERING - ";N
160  GOSUB 250
170  F=F+5.0E-03:F=100*F:F=INT(F):F=F/100
180  F1=F1+5.0E-03:F1=100*F1:F1=INT(F1):
     F1=F1/100
190  I=I1:K=K+1
200  IF ABS(F)-E>0 THEN 160
210  PRINT " "
220  X=Z*I:PRINT "THE EFFECTIVE INTEREST RATE IS
     ";100*X;" %"
230  PRINT " "
240  END
250  T=(1+I)^N
260  F=C-R/T-A*(1-1/T)/I
270  T2=T*(1+I)
280  F1=R*N/T2+A*(1-1/T-I*N/T2)/I/I
290  I1=I-F/F1
300  RETURN
```

NUMBER GAME

This program is a self-prompting number game. Enter a number, then the computer chooses a number between your number and zero. You then have the chance to guess the computer's number.

```
 10  'A make it easy or hard on yourself game, by
     Rich Lauck.
 20  fullw 2:clearw 2
 30  gotoxy 0,0
 40  ? " Let's play GUESS MY NUMBER."
 50  ? " You enter a number and press "
 60  ? " Return. Then I'll pick a number"
 70  ?" between your number and ";
 80  ?"zero."
 90  ? " Go ahead, enter a number "
100  INPUT " and press Return. ",TOP
110  ?:? "And now try to, GUESS MY NUMBER "
120  RANDOMIZE 0
130  ANSWER =INT(RND*(TOP))
140  ?:? " You guess and I'll give hints.":goto 180
150  ?:input " Y to play again, any other to quit. ",
     again$:?
160  if again$="Y" or again$="y" then 90
170  end
180  input guess
190  if guess < answer then ?"To low try
     higher.":goto 180
200  if guess > answer then ?"To high try
     lower.":goto 180
210  ? "You got my number.":goto 150
```

BOX DEMO

This low-resolution color program uses the AES and VDI to draw multi-colored boxes on a screen location of your choice.

AES (Applications Environment Services) is the part of GEM that allows for drop-down menus, multiple windows, and Dialog Boxes. VDI (Virtual Device Interface) is the part of GEM that contains graphics and text routines.

Follow these steps to use the program:

1. RUN the program.
2. With the mouse, point to the location on the screen where you want to draw the box.
3. Press the right mouse button to draw the box.
4. Press the left mouse button to exit from the program.

```
   5  a# = 9b
  10  control = peek (a#)
  20  global = peek(a# + 4)
  30  gintin = peek(a# + 8)
  40  gintout = peek(a# + 12)
  50  addrin = peek(a# + 16)
  60  addrout = peek(a# + 20)
 100  clearw 2:fullw 2
1070  poke systab+24,1
1071  poke contrl,122:poke contrl+2,0:poke con-
      trl+6,1
1072  poke intin,0:vdisys(1)
1074  mouse =1
1075  gemsys(79)
2000  x = peek(gintout + 2)
2010  y = peek (gintout + 4)
2020  key = peek (gintout + 6)
2025  if key = 2 then gosub 3000
2027  if key = 1 then poke systab+24,0:end
2028  if key = 0 then gosub 3115
2030  goto 1075
3000  rem
      ****************************************
3010  rem draw a box using vdi
```

```
3020  rem
      ****************************************
3022  color 1,(rnd*15)+1,1,rnd*25,2
3024  if mouse=0 then 3040
3030  mouse=0
3035  poke contrl,123:poke contrl+2,0:poke con-
      trl+6,0
3037  vdisys(1)
3040  poke contrl,11
3050  poke contrl + 2,2
3060  poke contrl + 6,0
3070  poke contrl + 10,1
3080  poke ptsin,x
3090  poke ptsin + 2,y
3095  poke ptsin + 4,x+50
3100  poke ptsin + 6,y+50
3110  vdisys(1)
3112  return
3115  if mouse=1 then return
3116  poke contrl,122:poke contrl+2,0:poke
      contrl+6,0
3117  poke intin,0:vdisys(1)
3120  mouse =1: return
3130  end
```

APPENDIX I
GLOSSARY

AES Applications Environment Services. The part of GEM that allows drop-down menus, multiple windowing, and Dialog Boxes. See GEM.

Alphanumeric The alphabetic letters A-Z and a-z, the numbers 0-9, and some symbols. Does not include punctuation marks or graphics symbols.

Array A list of numerical values stored in a series of memory locations. Arrays of more than 10 elements must be set up with a DIM statement.

ASCII American Standard Code for Information Interchange. A numeric code used to represent letters, numbers, and other symbols.

Assembler Routine See Machine Language Routine.

BASIC Beginner's All-purpose Symbolic Instruction Code. A high level programming language developed by Kemeny and Kurtz at Dartmouth College in 1963.

Binary A number system using base two. The only possible digits are 0 and 1, which may be used in a computer to represent true and false, on and off.

Bit Short for Binary Digit. The smallest unit of data with which a computer can work. A bit may be used to represent true or false, whether a circuit is on or off, or any other type of either/or concept.

Bug A mistake or error usually in the software of a program.

Byte Usually eight bits (enough to represent the decimal number 255 or 11111111 in binary notation). A byte of data can be used to represent an ASCII character or a number in the range of 0 to 255.

Code Instructions written in a language understood by a computer.

Command Mode An instruction to ST BASIC that is executed immediately. An example is the ST BASIC command RUN. See Statement.

Concatenation The process of joining two or more strings together to form one longer string. ST BASIC uses the plus sign (+) to concatenate strings.

Constant A value not contained in a variable. A constant is stated explicitly by its existence.

Cursor A square, rectangle, or vertical line displayed on the video display screen that shows where the next typed character will appear. If you move the mouse controller, an arrow-shaped cursor also appears, and remains until you use the keyboard again.

Data Information of any kind.

Debug The process of locating and correcting mistakes and errors in a program.

Default A mode or condition assumed by the computer until it is told to do something else. Default input and output devices are the keyboard and screen, which ST BASIC uses for INPUT and PRINT statements unless told to use other I/O devices.

Device Usually a piece of hardware (also known as a peripheral) used by a computer for input and/or output. Common examples of devices are printers, disk drives, and monitor screens.

Dialog Box Appears on the screen if an error condition occurs, such as trying to save a program on a nonexistent disk. Text in the box prompts your next action.

Direct Mode Refers to instructions to the computer that are executed immediately upon entry. See Command.

Editing Making corrections or changes in a program or data.

Error Messages Appear in the Command Window when something is wrong. You can create your own Error Messages with the ERROR statement.

Execute To do what a program or command specifies. To RUN a program or portion thereof.

Expression A combination of variables, numbers, and operators that can be evaluated to a single quantity. The quantity may be a string or a number.

Format To specify the form in which something is to appear.

GEM Graphics Environment Manager. GEM is the part of TOS that contains the VDI and AES.

Hard Copy Printed output as opposed to temporary screen display.

Increment Increase in value (usually) by adding one. Often used for counting (as in the number of repetitions through a loop). Opposite of decrement.

Indirect Mode The instructions to the computer that are contained within a program. Indirect statements are not executed immediately upon entry. See Statement.

Initialize Set to an initial or starting value. In ST BASIC, all non-array variables and strings are initialized to zero when the command RUN is given. Array elements, both string and numeric, are not initialized.

Input Information transfer to the computer. Input can come from a mouse, joystick, keyboard, digitizer, disk drive, and other devices.

Integer A number between -32768 and 32767, represented internally with two bytes. ST BASIC performs arithmetic faster with integers than with real numbers.

Interface The electronics used to allow two devices to communicate. Also used to describe the part of a program with which the user interacts.

I/O Short for input/output. I/O is used to describe any communication to or from the computer.

K Kilo meaning "times 1000." One Kbyte is actually 1024 (2 to the tenth power) bytes.

Keyword A word that has meaning as a command, function, or statement in a computer language, and must not be used as a variable name or at the beginning of a variable name.

Language A set of conventions specifying how to tell a computer what to do.

Logical Operator Used to control a program's decision-making ability. Logical operators work on the flags resulting from logical expressions. See, Appendix B.

Machine Language Routine A program written in machine language (the computer's most fundamental language) that can be used by an ST BASIC program with the CALL statement. Often used where speed is paramount.

Memory The part of a computer (usually RAM or ROM) that stores data or information.

Menu A list of options from which the user may choose.

Microcomputer A computer based on a microprocessor chip. The ATARI ST uses a Motorola 68000.

Null String A string containing no characters at all. If you use the INPUT statement to accept a string from a user at the keyboard, and the user only presses the **[Return]** key, a null string is returned.

Operator A symbol that permits arithmetic or logical manipulation of data. Examples of operators are + - * / > < .

OS Operating System. A collection of programs that allows the user to control the computer. The ATARI ST OS is TOS, "The Operating System".

Output Information transfer away from the computer. Examples of output devices are monitor, printer, and disk drive.

Parameters Quantities passed between a program and a routine; usually presented in a list of variables and/or constants separated with commas.

Parallel Two or more things occurring simultaneously. A parallel interface controls a number of distinct electrical signals at the same time. See Serial.

Peripheral An I/O device. See Device.

Pixel Picture Element. One point in the screen display. Pixel size depends on the screen resolution mode being used.

Precedence Rules that determine the priority in which operations occur, especially with regard to the arithmetical/logical operators.

Program A sequence of instructions to the computer. A program must be written in a language that the particular computer can understand.

Prompt A symbol that appears on the video display screen that indicates the computer is ready to accept keyboard input. In ST BASIC, this takes the form of the word Ok. A ? may also be used by your program to prompt a user to enter (input) information or take other appropriate action.

RAM Random Access Memory. The main memory in most computers. RAM is used to store both programs and data.

Random File A disk file whose records may be accessed in any order. In order to make efficient use of random files, your program must maintain a separate index of such files.

Record An item of data in a random or sequential disk file.

Reserved Word See Keyword.

ROM Read Only Memory. Contains information stored by the manufacturer that cannot be changed by the user.

Save To copy a program or data into some location other than RAM, usually to a disk drive.

Screen The video display screen.

Sequential File A disk file containing data that must be accessed in sequence. For example, in order to read the fifth record in a sequential file, you must first read the first through the fourth records.

Serial Refers to things happening one at a time in sequence. A serial interface, such as the RS232C port used by the ST for telecommunications, passes each byte of information one bit at a time. See Parallel.

Stack A LIFO (last-in, first-out) structure in the computer's memory used for temporary storage and quick retrieval of data. Often compared to a cafeteria spring-loaded dish stacker.

Statement An instruction to the computer, usually contained within a program. A statement must contain a line number, at least one keyword, and usually a value to be operated on. See Command.

String A sequence of characters that may contain letters, numbers, and punctuation, and begins and ends with a quotation sign. A string may be stored in a string variable, which usually ends in $.

Subroutine A part of a program to which the main program can branch (jump) and return many times. Subroutines permit the programmer to save memory by reusing the same routine without having to repeat it in the program. Subroutines are very powerful, and are used at just about every level of programming.

Variable A variable may be thought of as a box in which a value may be stored. Such values are typically numbers and strings.

VDI Virtual Device Interface. That part of GEM that contains graphics and text routines.

Window A portion of the monitor display devoted to a specific purpose. ST BASIC uses four variable-size windows; one each for Listing, Editing, Output, and Commands.

TOS See Operating System.

CUSTOMER SUPPORT

Atari Corp. welcomes any questions you might have about your ATARI Computer product.

Write to:

Atari Customer Relations
P.O. Box 61657
Sunnyvale, CA 94088

Please write the subject of your letter on the outside of the envelope.

We suggest that you contact your local Atari User Groups. They are outstanding sources of information on how to get the most out of your ATARI Computer. To receive a list of Atari User Groups in your area, send a self-addressed, stamped envelope to:

Atari User Group List
P.O. Box 61657
Sunnyvale, CA 94088

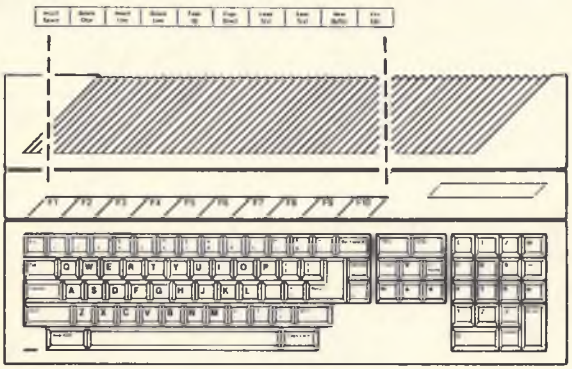
INDEX

FUNCTION KEYS TEMPLATE

With the function keys template attached to your ST keyboard, it will be easy to remember the editing function of each special function key.

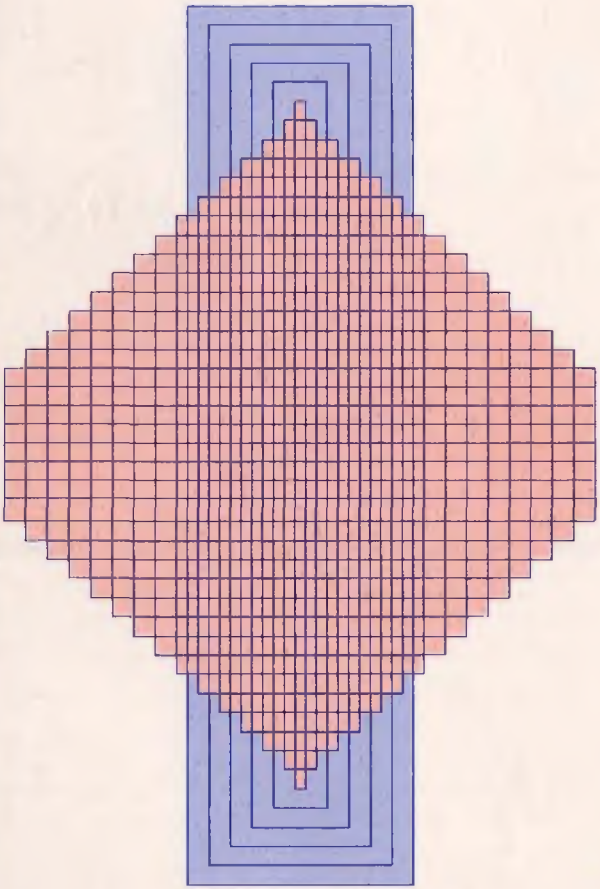
Assembling and attaching the template is simple, just follow these steps:

1. Tear out the two pieces of the template.
2. Tape the pieces together with clear tape.
3. Place on keyboard housing as shown in the illustration. Attach template to keyboard with clear tape.



Insert Space	Delete Char	Insert Line	Delete Line	Page Up

Page Down	Load Text	Save Text	New Buffer	Exit Edit



ATARI®

Atari Corp., Sunnyvale, CA 94086

CO26220 Rev. A
CO26166 1986 I C C