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 520ST™ Computer System.

The ST BASIC Sourcebook is set up for easy access to all the information a programmer needs. For the first-time BASIC programmer, work through the examples in Section 1 of the manual. The special characteristics of the language and the BASIC Desktop are demonstrated.

Section 2 is the reference section of the manual, containing comprehensive Appendices on every aspect of the language. A description of each reserved word, logical operators and order of precedence, Error Message listing, and sample programs are all provided.

Whether you are a beginning programmer or an expert, it is important that you make a backup copy of the ST Language disk before you begin programming. Refer to the 520ST Owner's Manual for detailed instructions on making a backup disk.

SECTION 1
GETTING STARTED WITH ST BASIC

The first section of this manual provides a general introduction to ST BASIC and demonstrates how BASIC works within the desktop environment of the ST Computer System.

This section is divided into three main parts:

*Loading ST BASIC

*Touring the ST BASIC Desktop

*Writing an ST BASIC Program

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 520ST 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.

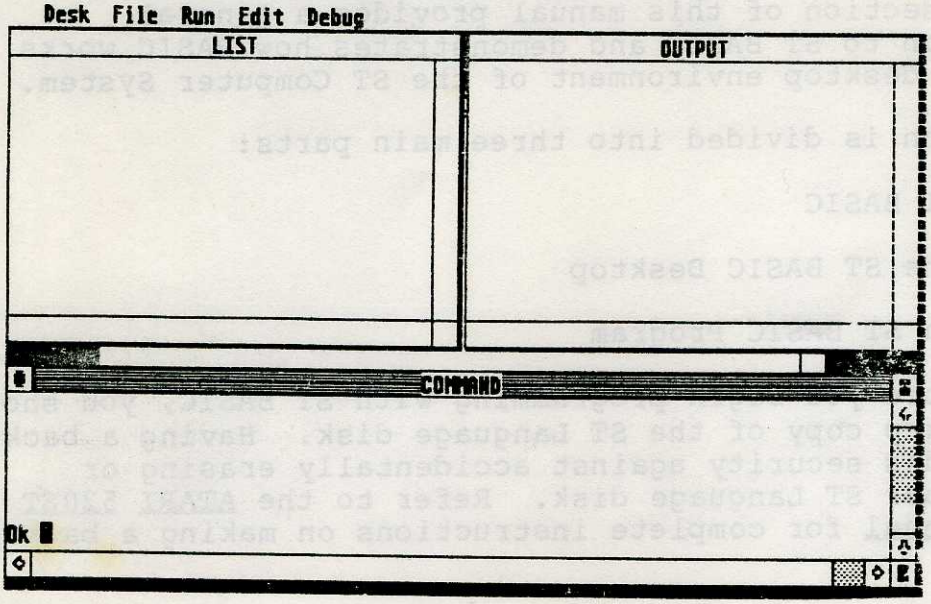
2. When the Dialog Box prompts you to insert Disk B into Drive A, place the ST Language disk into Drive A and press the [Return] key.

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



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

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 520ST Owner's Manual.

WINDOWS

The ST BASIC programming environment includes four windows: Command, Output, List, and Editor. After you load the ST BASIC program and the 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 520ST Owner's Manual. Please refer to that manual for specific information.

The Command Window

Enter ST BASIC commands and program lines 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 an abbreviation for the PRINT statement. Type the following program in the Command Window:

```
? 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, you should 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 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 A 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 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.

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 and click on the left mouse button. If the Exit button has an enlarged border, you can press the [Return] key on the ST 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, refer to Chapter 6 of the ATARI 520ST 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. The RENUM command is explained in detail in Appendix C.

Labels

ST BASIC also allows you to use labels to help identify program lines. 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: You can write ST BASIC programs in either all capital letters or upper- and lowercase letters.

ENTERING A PROGRAM

If there is anything in the List Window, clear it by typing

CLEARW 1

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 following program. 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.

Type [Control] [G] to stop the AUTO program.

This simple program illustrates a few ST BASIC features.

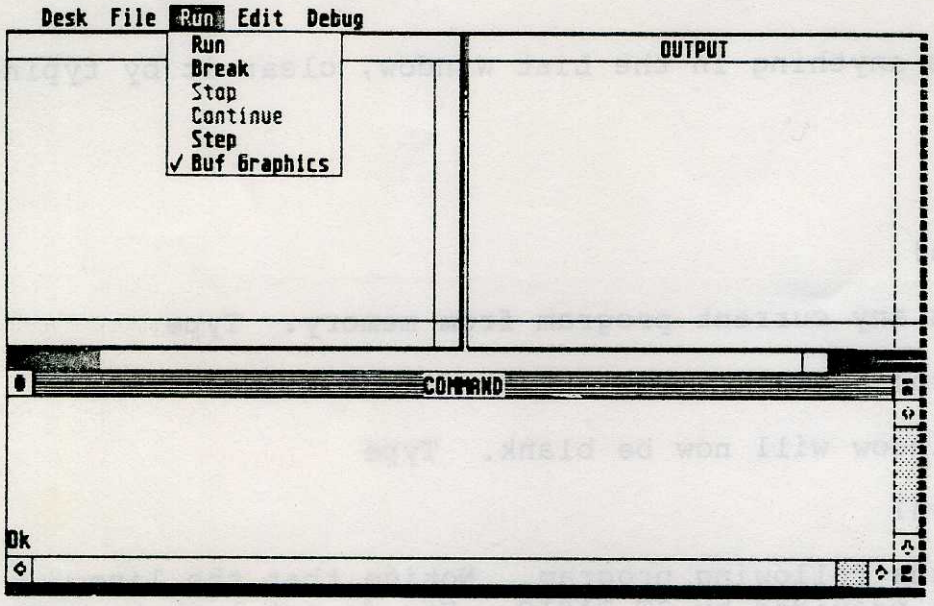
Line 10 has a REMark to help clarify its function. The REMark is ignored by ST BASIC. You can begin REMarks with a single quote ('), as in line 30.

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 first 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.

RUNNING A PROGRAM



Select the Run menu from the Menu Bar and then 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. When 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

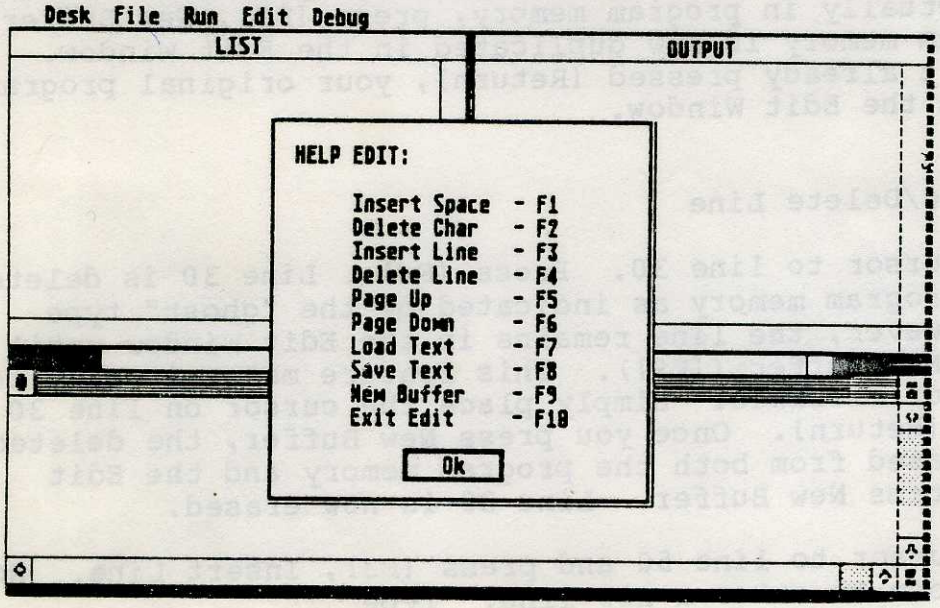
program line. To edit your program, select the Edit menu and click the Start Edit option (or type ED).

When you edit, you move the cursor to the place on the screen where you want to insert or delete a character, add or delete space. You control the cursor with the Cursor Control keys (arrow keys) on the ST keyboard.

Use the Cursor Control 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."

Function Keys

Before going further, 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 right.

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

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 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]

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

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.

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 type [Return] on 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 two lines toward the end of a program.

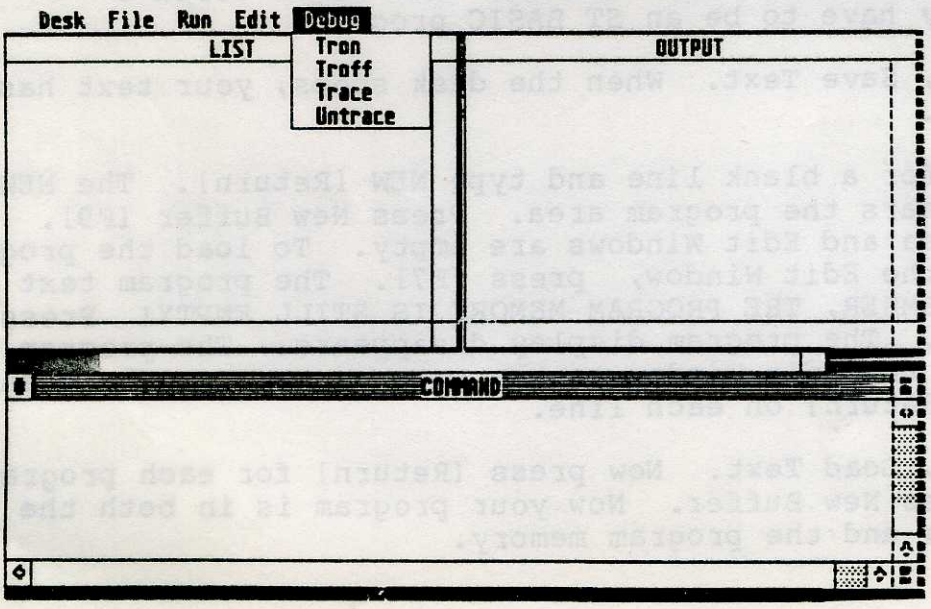
Note: The maximum visible line length is 80 characters. If you type off of the edge of the visible window, the text in the screen will move to the left so you can see what you're typing. You can type up to 80 characters in the Edit Window. If you attempt to edit a program with lines longer than 80 characters, 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).

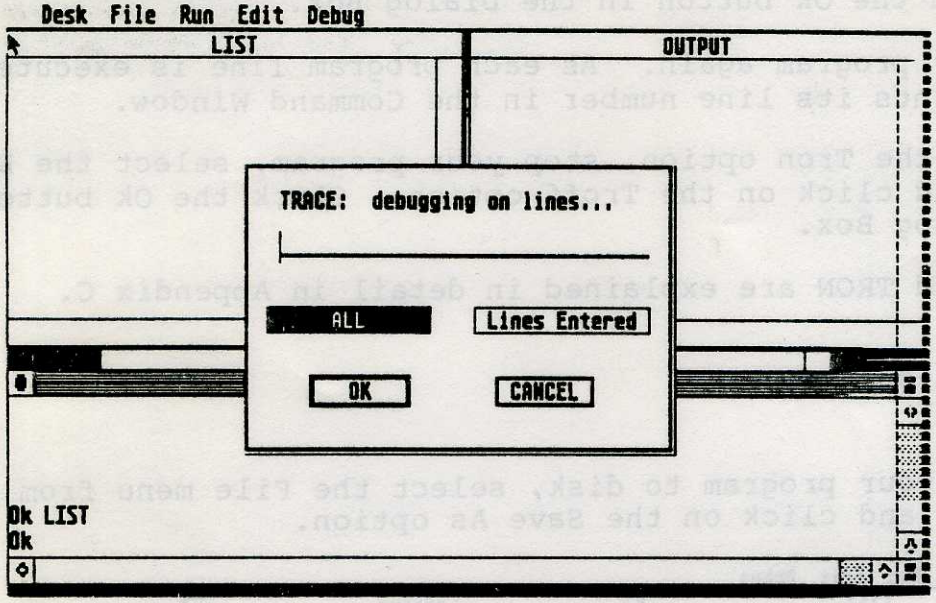
DEBUGGING A PROGRAM

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.

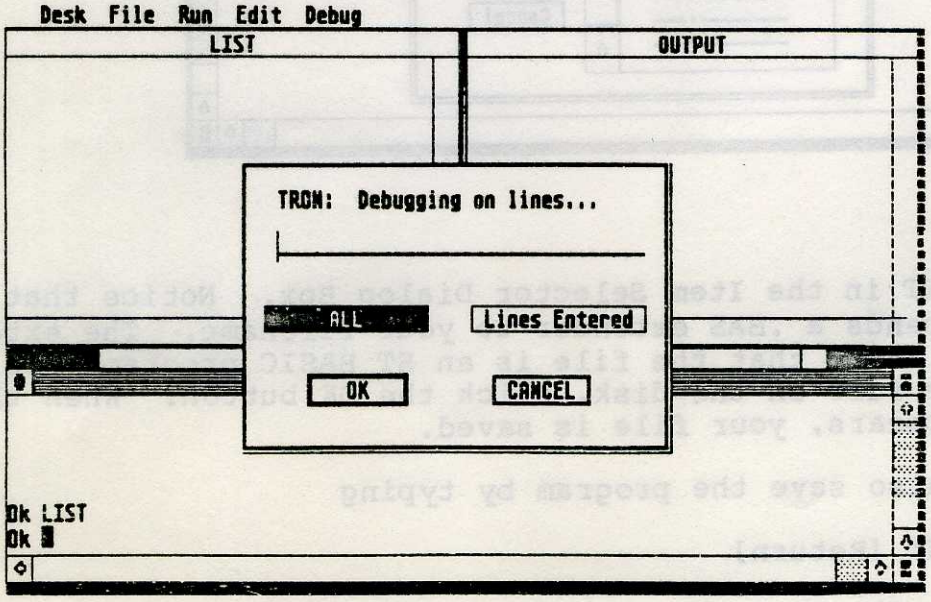


Click on the Ok button in the Dialog Box.

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.

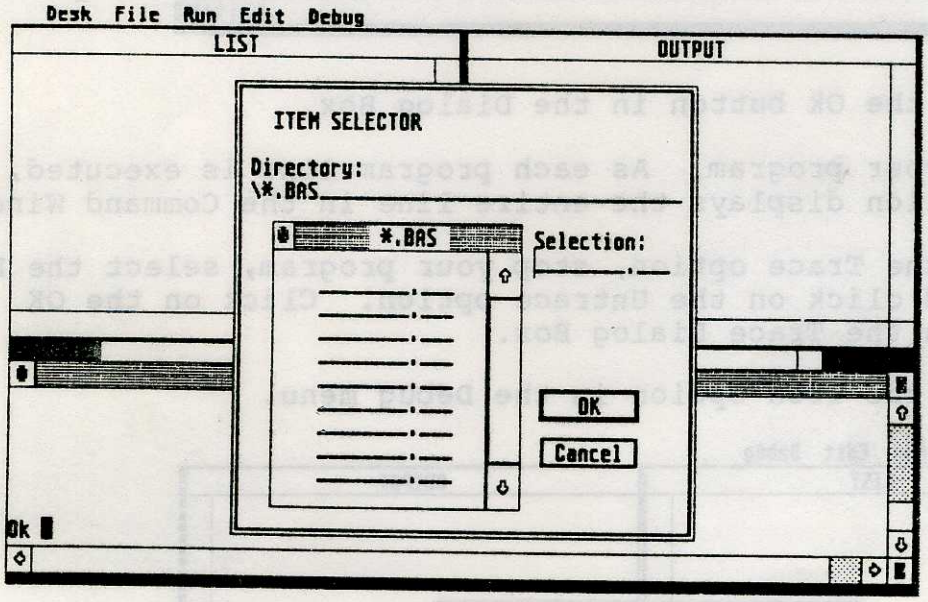
Run your program again. As each program line is executed, Tron prints its line number in the Command Window.

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.

SAVING A PROGRAM

To save your program to disk, select the File menu from the Menu Bar and 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 will not delete 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, click on the Delete File option in the File menu. 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.

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 or to stop AUTO line numbering)
[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
QUIT
RUN <filename>
SAVE <filename>
STEP
TRACE
TROFF
TRON
UNTRACE

A complete list of ST BASIC commands is in Appendix A of this manual.

BUFFERED GRAPHICS

To used buffered graphics with ST BASIC on the 520ST System, you must free some memory space.

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

1. Delete the desk accessories from the ST Language backup disk. Simply, open the ST Language 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 file. Select the 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 520ST Owner's Manual.

APPENDIX A
ST BASIC RESERVED WORDS

The following is a list of reserved words used in ST BASIC. If you use any of these words as a variable name, the Error Message, "Something is wrong," will appear on the screen. Each reserved word is explained in detail in Appendix C.

ABS
ASC
ATN
AUTO
BLOAD
BREAK
BSAVE
CALL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLEARW
CLOSE
CLOSEW
COLOR
COMMON
CONT
COS
CVD
CVI
CVS
DATA
DEF FN
DEF SEG
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DIR
EDIT
ELLIPSE
END
EOF
ERA
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILL
FIX
FLOAT
FOLLOW
FOR
FRE
FULLW
GEMSYS
GET
GOSUB
GOTO
GOTOXY
HEX$
IF
INP
INPUT
INPUT#
INPUT$
INSTR
INT
KILL
LEFT$
LEN
LET
LINE INPUT
LINE INPUT#
LINEF
LIST

LLIST
LOAD
LOC
LOF
LOG
LOG10
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKI$
MKS$
NAME
NEW
PEEK
PELLIPSE
POKE
POS
PRINT
PRINT#
PRINT USING
PUT
QUIT
RANDOMIZE
READ
REM
RENUM
REPLACE
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SGN
SIN
SOUND
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSTAB
SYSTEM
TAB
TAN
TRACE
TROFF
TRON
UNBREAK
UNFOLLOW
UNTRACE
VAL
VARPTR
VDISYS
WAIT
WAVE
WEND
WHILE
WIDTH
WRITE
WRITE#

APPENDIX B
LOGICAL OPERATORS, ORDER OF PRECEDENCE, AND ST BASIC FUNCTIONS

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. Thus, "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. Here, "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. Here, "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. Here, "A%=5: B%=3: C%=A% EQV B%" makes C% equal -7.

LOGICAL OPERATION 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
	Integer 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.

VARIABLE DECLARATION CHARACTERS

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

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.

NUMERIC FUNCTIONS

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 e 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.

STRING FUNCTIONS

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 A.M.	2 P.M.	10 P.M.	

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 section describes the ST BASIC commands, functions, and statements in alphabetical order. 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 X = ABS(N)

Syntax: X = ABS(<numeric expression>)

FUNCTION: Returns the absolute value of a number which is always positive or zero.

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 I% = ASC(A$)

Syntax: I% = ASC(<string expression>)

FUNCTION: Returns the ASCII value of the first character in a string.

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 the Appendices 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 ! = ATN(N%)

Syntax: X! = ATN(<numeric expression>)

FUNCTION: Returns the arctangent of a number.

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 AUTO

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

Syntax: AUTO [<starting line number>] [,<increment>]

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.

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:

```
Ok AUTO
10
20
30
 .
 .
 .

OK AUTO 50, 25
50
75
100
 .
 .
 .

Ok AUTO , 20
10
30
50
```

```
.
.
.

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

BLOAD BLOAD TESTFILE.DAT, 250

Syntax: BLOAD <filespec>[,<address>]

STATEMENT: Loads a file into memory.

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

Syntax: BREAK [<list of line numbers>]

COMMAND: Stops program execution.

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 BSAVE TESTFILE.DAT, 250, 500

Syntax: BSAVE <filespec>,<address>,<length>

STATEMENT: Saves part of memory to a file.

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 110 BSAVE "ARRAY" ,23,650
```

CALL CALL DRAW(X, Y, Z)

Syntax: CALL <numeric variable> [(<parameter list>)]

STATEMENT: Transfers control to a machine language subroutine.

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 8 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 8-byte value in the array will contain a pointer to the memory location containing that string.

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

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

STATEMENT: Transfers control and passes variables to another program. A .BAS extender is assumed unless otherwise specified.

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 file name.

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$ A$ = CHR$(97)

Syntax: A$ = CHR$(<numeric expression>)

FUNCTION: Returns the ASCII character that corresponds to the specified ASCII decimal value.

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 I% = CINT(N)

Syntax: I% = CINT(<numeric expression>)

FUNCTION: Rounds a number to the nearest integer.

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

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

STATEMENT: CIRCLE draws circles and arcs.

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. 0 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
```

Syntax: CLEAR

STATEMENT: Frees all memory used for program data without erasing the program currently in memory.

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
```

Syntax: CLEARW <numeric expression>

STATEMENT: CLEARW clears BASIC windows.

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 CLOSE

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

Syntax: CLOSE [#]<file number>

STATEMENT: Closes open disk files, concluding any input or output.

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. 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
```

Syntax: CLOSEW <window number>

STATEMENT: Close one basic window.

Explanation: Used to close one of four 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 BASIC interpreter that allow the system to keep track of the window status. Therefore, do not close basic windows using direct calls to AES.

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

STATEMENT: Sets text, fill, and plot colors and fill patterns.

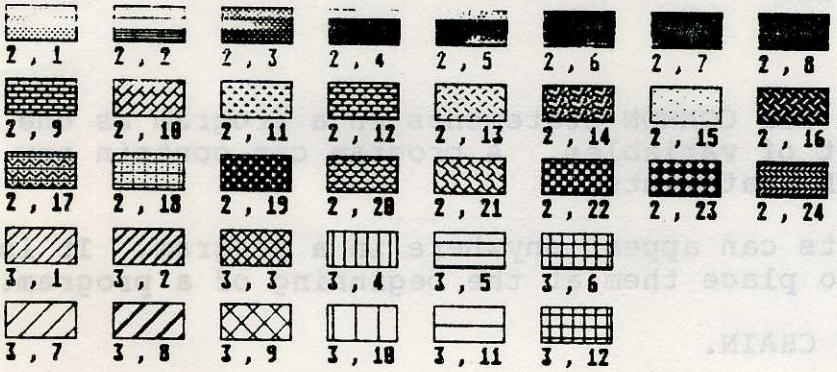
Explanation:

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 chart below shows the numbers for colors in different resolutions:

NUMBER	LOW	MED	HI
0	X	X	X
1	X	X	X
2	X	X	
3	X	X	
4	X		
5	X		
6	X		
7	X		
8	X		
9	X		
10	X		
11	X		
12	X		
13	X		
14	X		
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 left of the comma corresponds to the style: Hollow, Pattern, or Hatch. The number to the right 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"
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
```

Syntax: COMMON <variable>,<variable>

STATEMENT: Declares the variables that a program can pass to a chained program.

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"
```

Syntax: CONT

COMMAND: Resumes program execution from the BREAK mode.

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
Ok CONT
 1
 0
Ok
```

COS X = COS(Y)

Syntax: X = COS(<numeric expression>)

FUNCTION: Returns the cosine of a number.

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, and CVS

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

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

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

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 DATA 25,15,925,word

Syntax: DATA <constant>,<constant>

STATEMENT: Defines a list of constants that a READ statement can assign to variables.

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 DEF FNA(A) = A*2+5

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

STATEMENT: Defines user specified functions.

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;" INCH MATERIAL.":PRINT
"YOUR COST IS: ";FNCOST!
Ok 100 DEF FNINYARDS = FNSLACK / 36
```

```
Ok 110 PRINT FNSLACK; "INCHES 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 INCH MATERIAL.
YOUR COST IS 7.11111
 128 INCHES IN YARDS IS 3.55555
Ok
```

DEF SEG DEF SEG 0
         DEF SEG 1

Syntax: DEF SEG [<numeric expression>]

STATEMENT: DEF SEG establishes the mode of operation of PEEK and POKE, and the offset used by the commands.

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 DEFDBL A
          DEFDBL A-D

Syntax: DEFDBL <letter>-<letter>

STATEMENT: Declares a range of letters as defining real numbers.

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 DEFINT A
DEFINT A-D

Syntax: DEFINT <letter>-<letter>

STATEMENT: Declares a range of letters as defining integers.

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 DEFSNG A
        DEFSNG A-D

Syntax: DEFSNG <letter>-<letter>

STATEMENT: Declares a range of letters as defining real numbers.

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 DEFSTR A
DEFSTR A-D

Syntax: DEFSTR <letter>-<letter>

STATEMENT: Declares a range of letters as defining strings.

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

Syntax: DELETE <line number list>

COMMAND: DELETE erases program lines from memory.

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

Syntax: DIM<array name>(<subscript>,<subscript>)
(,<array name>[<subscript>])

STATEMENT: Defines the number of dimensions and the number of elements in an array.

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, etc. 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) = "FLOORPLAN3"
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 80 HOUSES$ (1,1,1) = "FLOORPLAN2"
Ok 90 IF HOUSES$ (1,0,0) = "FLOORPLAN2" THEN GOTO 300
```

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

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

COMMAND: Lists the files on a disk.

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. e.g., FIG.BAS, PIG.BAS, BIG.BAS, etc.)

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 EDIT ED
EDIT 30 ED 30

Syntax: EDIT <LINE NUMBER> ED <LINE NUMBER>

COMMAND: Invokes the ST BASIC editor.

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

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

STATEMENT: ELLIPSE draws ellipses and elliptical arcs.

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. 0 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:CLEARW2
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 END

Syntax: END

STATEMENT: Stops program execution, closes all files, and returns to command level.

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 X = EOF(1)

Syntax: X = EOF(<file number>)

FUNCTION: Returns true (-1) at the end of a sequential or random access file.

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 ERA MYFILE.TXT
       ERA B:MYFILE.TXT

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

COMMAND: Deletes a file from the disk.

Explanation:

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

ERASE ERASE A$, B$, C

Syntax: ERASE <array name>,<array name>

STATEMENT: Erases arrays.

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 X=ERL
X=ERR

Syntax: X=ERL
X=ERR

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

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 ERROR X

Syntax: ERROR<numeric expression>

STATEMENT: Simulates a BASIC run time error and transfers control to an error trapping routine.

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 X = EXP(Y)

Syntax: X = EXP(<numeric expression>)

FUNCTION: Returns the constant e raised to an exponent.

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 FIELD #1, 8 AS X$, 4 AS Y$, 2 AS S$

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

STATEMENT: Allocates variable space in random file buffers.

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
Ok 110 FIELD #5, 20 AS I$, 10 AS D$, 10 AS E$
```

FILL FILL 150,80

Syntax: FILL <numeric X expression>,<numeric Y expression>

STATEMENT: Fills shapes with colors or patterns.

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 X = FIX(Y)

Syntax: X = FIX(number)

FUNCTION: Truncates a real number to an integer.

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 X = FLOAT(Y)

Syntax: X = FLOAT(<integer expression>)

FUNCTION: Converts an integer to a real number.

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 = FLOAT(97)
Ok 20 PRINT X
Ok RUN
 97
```

FOLLOW FOLLOW N
          FOLLOW N, B

Syntax: FOLLOW <variable>[,<variable>]

COMMAND: Follows the values of program variables.

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 FOR I = 1 TO 5 STEP 1

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

STATEMENT: Creates a loop that executes a given number of times.

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 X = FRE(0)

Syntax: X = FRE(<dummy argument>)

FUNCTION: Returns the number of unused bytes in memory.

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 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.

Syntax: FULLW <numeric expression>

STATEMENT: Sets BASIC windows to full screen size.

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 GEMSYS(X)

Syntax: GEMSYS(<AES Op Code>)

FUNCTION: GEMSYS allows the user to access the operating system's AES interface.

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)
```

Syntax: GET [#]<file number> [,<record number>]

STATEMENT: Reads a record from a random disk file into a file buffer.

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 GOSUB 250
       GOSUB ENTRY

Syntax: GOSUB <line number> or GOSUB <label name>

STATEMENT: Passes program control to a subroutine.

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 GOTO 50
      GOTO ENTRY

Syntax: GOTO <line number> or GOTO <label name>

STATEMENT: Passes program control unconditionally to a given line number.

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 GOTOXY X,Y

Syntax: GOTOXY <Column Position>,<Row Position>

STATEMENT: Places output cursor at column and row position.

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$ X = HEX$(Y)

Syntax: X = HEX$(numeric expression)

FUNCTION: Returns a string that is the hexadecimal representation of a number.

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  IF X=Y THEN PRINT A: GOTO 250
      ELSE GOTO 30
```

Syntax: IF <logical expression> THEN <statement> <:statement>
[ELSE <statement> <:statement>]

STATEMENT: Sets conditions that determine program flow.

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 X = INP(3)

Syntax: X = INP(<port number>)

FUNCTION: Returns a byte value from a selected input port.

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 (Screen)
3 = MIDI (Musical Instrument Digital Interface)
4 = KEYBOARD
```

Example:

```
Ok 200 Y = INP(3)
Ok 210 IF INP(3) > X THEN GOTO 200
```

```
INPUT   INPUT A$
          INPUT "NAME: ", A$
          INPUT "NAME"; A$
          INPUT X, Y, Z
          INPUT "Height, Weight, Age", X, Y, Z
```

Syntax: INPUT [;] [<prompt string><; or,>] <variable>
,<variable>

STATEMENT: Lets you enter data while the program is running and assigns the data to program variables.

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# INPUT#1, A$, X

Syntax: INPUT#<file number>, <variable>, <variable>

STATEMENT: Reads data from a sequential disk file to program variables.

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$ X$ = INPUT$(6)
        X$ = INPUT$(6,#1)

Syntax: X$ = INPUT$(<number of characters>[,[#]<file number>])

FUNCTION: Returns a specified number of characters from the keyboard or a data file.

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 X = INSTR(3,A$,"DO")
       X = INSTR(3,A$,B$)

Syntax: X = INSTR([<starting point>,] <target string expression>, <pattern string>)

FUNCTION: Searches for one string within another and returns its position.

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 X = INT(Y)

Syntax: X = INT(numeric expression)

FUNCTION: Converts a number or expression to an integer.

Explanation:

INT truncates decimal places.

Example:

```
Ok 10 X = INT(2.999)
Ok 20 PRINT X
Ok RUN
 2
Ok
```

KILL KILL "FILE.DAT"

Syntax: KILL<string expression>

STATEMENT: Deletes a disk file.

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$ X$ = LEFT$(A$,5)

Syntax: X$ = LEFT$(<target string><number of characters>)

FUNCTION: Returns a string that contains the leftmost characters of a string.

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 Z = LEN(A$)

Syntax: Z = LEN(<string expression>)

FUNCTION: Returns the length of a string.

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

```
LET X(1)=Y
LET X=Y
```

Syntax: LET <variable>=<expression>
STATEMENT: Assigns a value to a variable or array variable.

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 LINE INPUT "NAME? "; A$
LINE INPUT; "NAME? "; A$

Syntax: LINE INPUT[;] [<prompt>[,or ;]]<string variable>

STATEMENT: Requests input from the keyboard and assigns it to a string variable.

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# LINE INPUT#1, A$

Syntax: LINE INPUT#<file number>, <string variable>

STATEMENT: Requests input from a sequential disk file and assigns it to a string variable.

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
```

Syntax: LINEF <point pair, point pair>

STATEMENT: LINEF draws a line.

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 portions]
 Ok
```

LIST LIST

```
LIST 10-50
LIST 10, 30, 50
LIST 10-30, 70-90
LIST - 30
```

Syntax: LIST [<line descriptor, list>]

COMMAND: Displays program lines in the LIST window.

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 LLIST
        LLIST 10-50
        LLIST 10, 30, 50
        LLIST 10-30, 70-90
        LLIST - 30

Syntax: LLIST [<line descriptor list>]

COMMAND: LLIST lists the program to your printer.

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 LOAD MYPROG

Syntax: LOAD <filename>

COMMAND: LOADs program files.

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 X = LOC(1)

Syntax: X = LOC(<file number>)

FUNCTION: Returns either a record number or the number of bytes read from or written to a file.

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 X = LOF(1)

Syntax: X = LOF(<file number>)

FUNCTION: Returns the number of bytes in the file.

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 X = LOG(N)

Syntax: X = LOG(<numeric expression>)

FUNCTION: Returns the natural logarithm of a number.

Explanation:

The numeric expression must be greater than zero.

Example:

```
Ok 10 PRINT LOG(23)/LOG(2)
Ok RUN
 4.52356
Ok
```

LOG10 X = LOG10(Y)

Syntax: X = LOG10(<numeric expression>)

FUNCTION: Returns the base 10 logarithm of a number.

Explanation:

The numeric expression must be greater than zero.

Example:

```
Ok 10 X = LOG10(1000)
Ok 20 PRINT X
Ok RUN
 3
```

Syntax: LPOS(X)

FUNCTION: Returns the position of the line printer print head within the line printer buffer.

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 LPRINT A$; " = "; X
LPRINT USING F$; A$, X

Syntax: LPRINT [<list of expressions>]
LPRINT USING <format string expression>;<list of expressions>

STATEMENT: Directs output to a printer.

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 LSET A$=B$

Syntax: LSET<string variable>=<string expression>

STATEMENT: Moves a string into a specified string variable without reassigning the string variable.

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 MERGE MYPROG

Syntax: MERGE <filename>

COMMAND: Inserts an ST BASIC disk file into a program in memory.

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$ X$ = MID$(A$,5,10)
      MID$(A$,5,5)= "HELLO"

Syntax: X$ = MID$(<string expression>, <starting point>,[<length>])

FUNCTION: Returns a segment of a string.

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$	X$ = MKD$(A)	A is a numeric value.
	X$ = MKI$(B)	B is an integer value.
	X$ = MKS$(C)	C is a numeric value.

Syntax: X$ = MKD$(<numeric expression>)
X$ = MKI$(<integer>)
X$ = MKS$(<numeric expression>)

FUNCTION: The MKD$, MKI$, and MKS$ functions convert ASCII strings representing numbers to byte strings for use in random file buffers.

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 NAME "AUG.DAT" AS "LAST.DAT"

Syntax: NAME <old string expression> AS <new string expression>

STATEMENT: Renames a file.

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 NEW NEWPROG.BAS

Syntax: NEW [NAME]

COMMAND: Clears a file from memory, and optionally names the new program.

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 NEXT X
        NEXT X, Y

Syntax: NEXT [<counter>] ,counter

STATEMENT: Marks the end of a FOR/NEXT loop.

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
X
Y
X
X
Y
X
X
Ok
```

OCT$ X$ = OCT$(Y)

Syntax: X$ = OCT$(<numeric expression>)

FUNCTION: Returns the string expression of an octal (base 8) number.

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
```

Syntax: OLD <filename>

COMMAND: Loads an existing program file into memory.

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 ON X GOTO INIT, 100, ENTRY, DONE
 ON X GOSUB INIT, 100, ENTRY, DONE

Syntax: ON <numeric expression> GOTO <line descriptor> [<line descriptor>]
ON <numeric expression> GOSUB <label> [,<label>]

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.

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 ON ERROR GOTO 200

Syntax: ON ERROR GOTO <line descriptor>

STATEMENT: Provides a mechanism to detect run time errors and pass control to a line number when an error occurs.

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  OPEN "O",#1,"FILE.DAT",128
        OPEN "I",#1,"FILE.DAT",128
        OPEN "R",#1,"FILE.DAT",128
```

Syntax: OPEN <mode>,[#]<file number>,<filename>[,<record length>]

STATEMENT: Lets you input and output to a file or device.

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 in the 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!
```

Syntax: OPENW <window number>

STATEMENT: Opens one ST BASIC window.

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 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 BASIC interpreter that allow the system to keep track of the window status. Therefore, do not open BASIC windows (that were closed using CLOSEW) using direct calls to AES.

OPTION BASE OPTION BASE 0
OPTION BASE 1

Syntax: OPTION BASE <1 or 0>

STATEMENT: Sets the base for array dimensions.

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 OUT 2,X

Syntax: OUT <integer expression>,<integer expression>

STATEMENT: Sends a byte to an output port.

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 (Screen)
3 = MIDI (Musical Instrument Digital Interface)
4 = KEYBOARD
```

Example:

```
Ok 100 IF X%>5 THEN OUT 3,(X-2)
```

PCIRCLE PCIRCLE 50,80,50
PCIRCLE 50,80,50,900,1800

Syntax: PCIRCLE <horizontal center,vertical center,radius>[<,start angle,end angle>]

STATEMENT: PCIRCLE draws solid circles and pie shapes.

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 X = PEEK(Y)

Syntax: X = PEEK(<memory location>)

FUNCTION: Returns the content of a memory location.

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 DEFBDL, 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 PELLIPSE 50,80,100,50
          PELLIPSE 50,80,100,50,900,1800

Syntax: PELLIPSE <horizontal center,vertical center,horizontal radius,vertical radius>[<,start angle,end angle>]

STATEMENT: PELLIPSE draws SOLID ellipses and elliptical pie shapes

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 POKE 1565,X

Syntax: POKE<location to poke>,<data to poke>

STATEMENT: Writes data to POKE to the memory.

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 X = POS(0)

Syntax: X = POS(<dummy argument>)

FUNCTION: Returns the current position of the cursor on the screen or printer.

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 PRINT X,Y
        PRINT X;Y
        Print A$
        ?A$
```

Syntax: PRINT [<expression><, or ;><expression>[<, or ;>]]

STATEMENT: Prints data to the output window.

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# PRINT# 1,A$,X
          ?#

Syntax: PRINT# <file number>,<expression>,<expression>

STATEMENT: Outputs data to a disk file.

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 PRINT USING FORM$;X,Y,Z
PRINT# 1,USING FORM$;X,Y,Z
?USING

Syntax: PRINT USING<string expression>;<list of expression">;
PRINT#<filenumber>,USING<"string expression">;<list of expressions>

STATEMENT: Prints output according to a format.

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 table summarizes 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
```

Syntax: PUT [#]<file number>,<record number>

STATEMENT: Writes a record from a buffer to a random disk file.

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%
```

Syntax: QUIT

COMMAND: Leaves ST BASIC and returns to GEM.

Explanation:

QUIT closes all files and returns you to GEM command level. Any program in memory is lost.

Same as: SYSTEM

Example:

```
Ok QUIT
```

Syntax: RANDOMIZE [<numeric expression>]

STATEMENT: Seeds the random number generator.

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 READ A,B,A$

Syntax: READ<variable>,<variable>

STATEMENT: Assigns values from a DATA statement to variables.

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 REM THIS IS A REMARK
       ' THIS IS A REMARK

Syntax: REM <remark>

STATEMENT: Introduces 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
.
.
.
```

Syntax: RENUM [<new first line>][,<starting line>][,<increment>]

STATEMENT: Renumbers program lines.

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 soon.

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 REPLACE MYPROG.BAS
         REPLACE MYPROG.BAS, 100-800

Syntax: REPLACE [<filename>][,<line number list>]

STATEMENT: Replaces an old version of a file with a new version.

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
```

Syntax: RESET

STATEMENT: RESET places the contents of the output window into the graphics buffer.

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 RESTORE 200

Syntax: RESTORE <line descriptor>

STATEMENT: Rereads DATA statements.

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 RESUME (0)
          RESUME NEXT
          RESUME 200

Syntax: RESUME (0)
        RESUME NEXT
        RESUME <line descriptor>

STATEMENT: Continues execution after an error.

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
```

Syntax: RETURN

STATEMENT: Transfers control from a subroutine to the statement following the last GOSUB.

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$ X$ = RIGHT$(A$,5)

Syntax: X$ = RIGHT$(<target string>, <number of characters>)

FUNCTION: Returns the rightmost characters of a string.

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 X = RND
       X = RND(Y)
       X = RND(0)
       X = RND(-Y)

Syntax: X = RND[(<numeric expression>)]

FUNCTION: Generates and returns a random number.

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 RSET A$=B$

Syntax: RSET<string variable>=<string expression>

STATEMENT: Moves a string into a specified string variable without reassigning the string variable.

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 RUN
       RUN ,200
       RUN MYPROG.BAS

Syntax: RUN
        RUN <,line descriptor>
        RUN <filename>

COMMAND: Begins program execution.

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 SAVE MYFILE
      SAVE MYFILE, 20-30
      SAVE MYFILE, 10, 30, 70, 80
      SAVE MYFILE, -30

Syntax: SAVE [<filename>], [<line descriptor list>]

COMMAND: Saves program lines to disk.

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 X = SGN(Y)

Syntax: X = SGN(<numeric expression>)

FUNCTION: Returns the sign of a number.

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 X = SIN(Y)

Syntax: X = SIN(<numeric expression>)

FUNCTION: Returns the sine of its argument expressed in radians.

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 SOUND VOICE, VOLUME, NOTE, OCTAVE, DURATION

Syntax: SOUND <numeric expression>, <numeric expression>, <numeric expression>, <numeric expression>, <numeric expression>,

STATEMENT: SOUND controls the 3 sound channels.

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$ X$ = SPACE$(Y)

Syntax: X$ = SPACE$(<numeric expression>)

FUNCTION: Returns a string of spaces.

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); "|"
Ok 40 NEXT V
Ok 50 FOR Z = 1 TO 21
Ok 60 PRINT "-";
Ok 70 NEXT Z
Ok RUN
              |
              |
              |
              |
              |
   ---------------------
```

SPC PRINT SPC(X)

Syntax: PRINT SPC(<numeric expression>)

FUNCTION: Outputs spaces to a PRINT statement.

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 X = SQR(Y)

Syntax: X = SQR(numeric expression)

FUNCTION: Returns the square root of a number

Explanation:

The number must not be negative. SQR returns a real number.

Example:

```
Ok 10 PRINT SQR(9)
Ok RUN
 3
Ok
```

STEP STEP
        STEP, 200
        STEP MYPROG.BAS

Syntax: STEP
        STEP <,line descriptor>
        STEP <filename>

COMMAND: Executes a program line by line.

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 STOP

Syntax: STOP

STATEMENT: Stops program execution, and transfers the control of BASIC to the Command Window.

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$ X$ = STR$(Y)

Syntax: X$ = STR$(<numeric expression>)

FUNCTION: Returns a string containing the decimal character representation of its argument.

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$ X$ = STRING$(Y,A$)
X$ = STRING$(Y,N)

Syntax: X$ = STRING$(<numeric expression>, <numeric or string expression>)

FUNCTION: Returns a string of a given length. The characters are defined by the second argument.

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
```

Syntax: SWAP <first variable>,<second variable>

STATEMENT: Trades the values of two variables.

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 X = PEEK(SYSTAB+OFFSET)

Syntax: X = PEEK(SYSTAB+OFFSET)

VARIABLE: System pointer table.

Explanation:

SYSTAB is the beginning memory location of a table of system parameters and pointers. 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. (See Table Below.)
*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 BASIC is off, no 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.

SYSTEM SYSTEM

Syntax: SYSTEM

COMMAND: Leaves ST BASIC and returns to GEM.

Explanation:

SYSTEM closes all files and returns you to GEM command level. Any program in memory is lost.

Same as: QUIT

Example:

```
Ok SYSTEM
```

Syntax: PRINT TAB(<tab position>)

FUNCTION: Moves the cursor to a specified tab position.

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 X = TAN(Y)

Syntax: X = TAN(<angle in radians>)

FUNCTION: Returns the tangent of a number.

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 30 PRINT TANGENT!
RUN
-.6235
Ok
```

TRACE TRACE
         TRACE 20,40
         TRACE 20-40
         TRACE -40

Syntax: TRACE [<line descriptor list>]

COMMAND: Follows program execution line by line and selectively prints the entire line.

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 TROFF

```
TROFF 10, 40
TROFF 10-40
TROFF -40
```

Syntax: TROFF [<line descriptor list>]

COMMAND: Cancels the TRON command.

Explanation:

TROFF cancels TRON either completely or for selected lines.

See: TRON

TRON TRON
        TRON 20,40
        TRON 20-40
        TRON -40

Syntax: TRON [<line descriptor list>]

COMMAND: Selectively traces program execution line by line and prints the line numbers.

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 UNBREAK

```
UNBREAK 20, 50
UNBREAK -50
UNBREAK 20-50
```

Syntax: UNBREAK [<line, descriptor list>]

COMMAND: Selectively cancels a BREAK command.

Explanation:

UNBREAK cancels BREAK either completely or for selected lines.

See: BREAK

UNFOLLOW UNFOLLOW
          UNFOLLOW X, Y

Syntax: UNFOLLOW [<variable>],[<variable>]

COMMAND:

Explanation:

UNFOLLOW cancels FOLLOW either completely or for selected variables.

See: FOLLOW

UNTRACE UNTRACE

```
UNTRACE 10, 40, 70
UNTRACE 10-40
UNTRACE -40
```

Syntax: UNTRACE [<line descriptor list>]

COMMAND: Cancels the TRACE command.

Explanation:

UNTRACE cancels TRACE either completely or for selected lines.

See: TRACE

VAL X = VAL(A$)

Syntax: X = VAL(<digit string expression>)

FUNCTION: Scans a string of characters and converts them to a real number.

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 X = VARPTR(Y)
          X = VARPTR(#1)

Syntax: X = VARPTR(<variable>)
        X = VARPTR(#<file number>)

FUNCTION: Returns the address of a variable.

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)
```

Syntax: VDISYS(<Dummy Argument>)

FUNCTION: Allows user to access the operating system's VDI interface.

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)
```

Syntax: WAIT <port number>,<integer expression>[,<integer expression>]

STATEMENT: Halts the program while waiting for an I/O port to develop a bit pattern.

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 WAVE ENABLE, ENVELOPE, SHAPE, PERIOD, DELAY

Syntax: WAVE <numerical expression>, <numerical expression>, <numerical expression>, <numerical expression>, <numerical expression>,

STATEMENT: WAVE controls the waveforms used in SOUND statements.

Explanation:

ENABLE is the mixer register of the sound generator. A 0 on bits 0-2 enable 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 below.

PERIOD sets the period of the envelope.

DELAY sets the time in 1/50 second increments before BASIC resumes execution.

Syntax: WEND

STATEMENT: Signals the end of a WHILE/WEND loop.

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
```

Syntax: WHILE <logical expression>

STATEMENT: States a condition that controls a WHILE/WEND loop.

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 WIDTH 72
WIDTH LPRINT 72

Syntax: WIDTH [LPRINT] <integer expression>

STATEMENT: Sets the line width of the screen or printer.

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, 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 WRITE X,Y,A$

Syntax: WRITE[<expression>],<expression>

STATEMENT: Outputs data to the terminal.

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# WRITE #1,X,Y,A$

Syntax: WRITE#[<expression>],<expression>

STATEMENT: Outputs data to a sequential file.

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 KILOWATT 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	Not used.
22	Expression has operator with no following operand.
23	Program line too long.
24-29	Not used.
30	Window number invalid.
31	Argument out of range.
32	Command cannot be executed from the editor.
33	Line is too complex.
34-49	Not used.
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	Not used.
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.

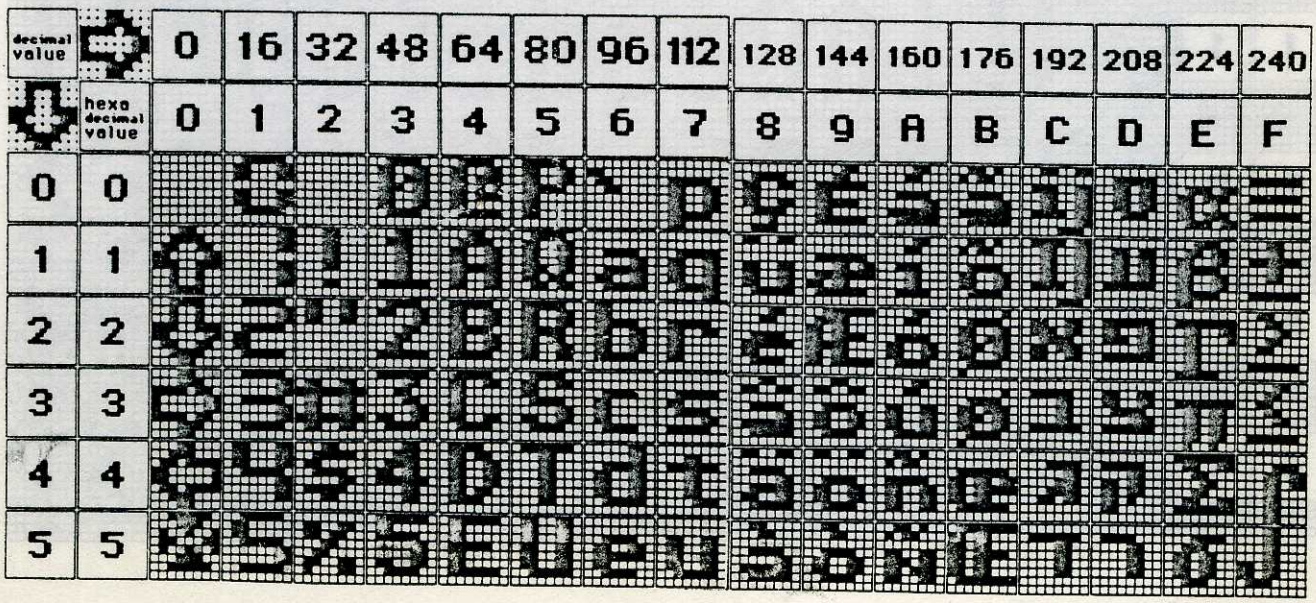
66	Program file has statement with no line number.
67-98	Not used.
99	--Break--.
100	Undefined error.
101	Program has too many lines.
102	ON statement is out of range.
103	Invalid line number.
104	A variable is required.
105	Undefined error.
106	Line number does not exist.
107	Number too large for an 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	Not used.
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	Not used.
221	System error #X, please restart.
222	Program not run.
223	Too many FOR loops.

APPENDIX E
ST ASCII CHARACTER SET

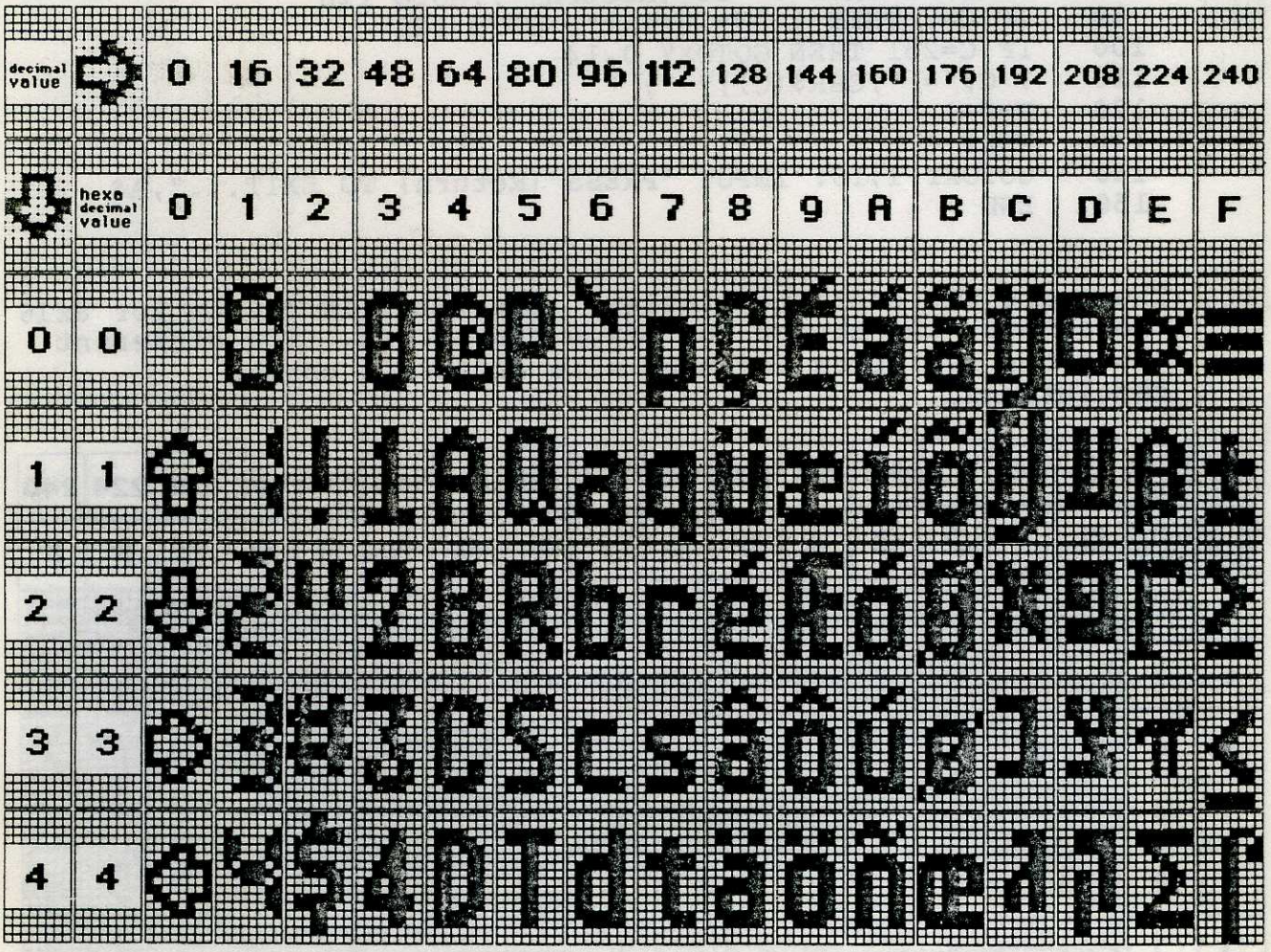
The following tables show the complete character sets available on the ST Computer. To print any of these characters from ST BASIC, input and run the following program:

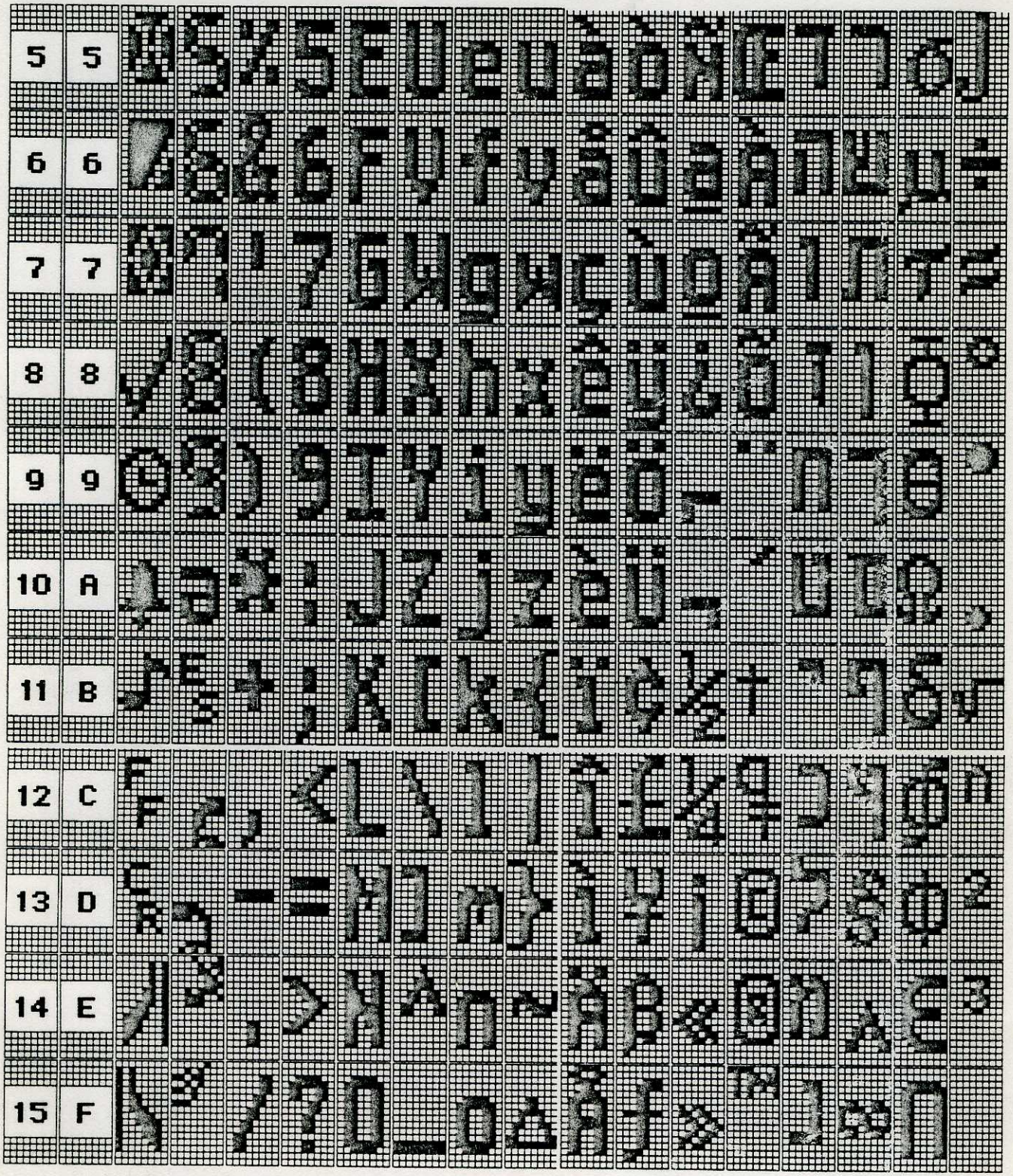
```
  5   ' THIS PROGRAM PRINTS A LIST OF ALL ST ASCII
      CHARACTERS  .
  6   ' AND THEIR CODES.
 10   FULLW 2:CLEARW 2
 20   GOTOXY 1,2:?"LIST OF ST ASCII CHARACTERS":GOTOXY 0,4
 30   P=0:1=0
 40   FOR C=1 TO 255
 50   IF P>4 THEN P=0:1=1+1:?
 60   IF 1=10 THEN GOTOXY 1,14: INPUT "pRESS [Return] TO
      CONTINUE...",A$
 70   IF 1=10 THEN 1=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 8x16 characters; the second for 16x16 characters. The different character set sizes are used with different screen resolutions.




6 6
7 7
8 8
9 9
10 A
11 B
12 C
13 D
14 E
15 F


decimal value
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240
hexa decimal value
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

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 BASIC (passing any necessary parameters to it).

Parameters are passed from 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 BASIC, the Assembler program can put any parameters it wants to pass to BASIC into a given memory location. Later, the 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 expresson to be evaluated.

APPENDIX H
SAMPLE PROGRAMS

BOXES

An interesting example of the RND statement using color graphics. Run this program in low-resolution 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=306: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
160    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
580   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=1:xl=1:yl=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 xl,yl,x,y
320    SKIP: xl=x
330    yl=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. You enter a number, the computer chooses a number between your number and zero, and then you 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# = gb
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 contrl+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 contrl+6,0
3037  vdisys(1)
3040  poke contrl,11
3050  poke contrl + 2,2
3060  poke contrl + 6,0
```

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

ST BASIC SOURCEBOOK

ERRATA SHEET

Enhancing ST BASIC's Memory

After you load TOS from the System Disk and ST BASIC from the ST Language disk, you will have a limited amount of memory space available for programming.

There are two methods to enhance the available memory space:

1. Turning off the Buffer Graphics option will provide an additional 32,000 bytes of memory. Point at the Run Menu on the 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.

2. Disabling the GEM desk accessories will provide 30,000 additional bytes of memory. Refer to page 17 of the ST BASIC Sourcebook for instructions.

Reserved Word lists

The following words should be added to the reserved word list in the manual:

ALL
AND
AS
BASE
CDBL
CONTRL
CSNG
DEF
ELSE
EQV
FIELD#
GB
GET#
GO
IMP
INKEY$
INTIN
INTOUT
MOD
NEXT
NOT
OCT$
OLD
ON
OPEN
OPENW
OPTION
OR
OUT
PCIRCLE
PTSIN
PTSOUT
SYSDBG
THEN
TO
USING
XOR

ATARI®

Printed in UK
CO26166 Rev. A