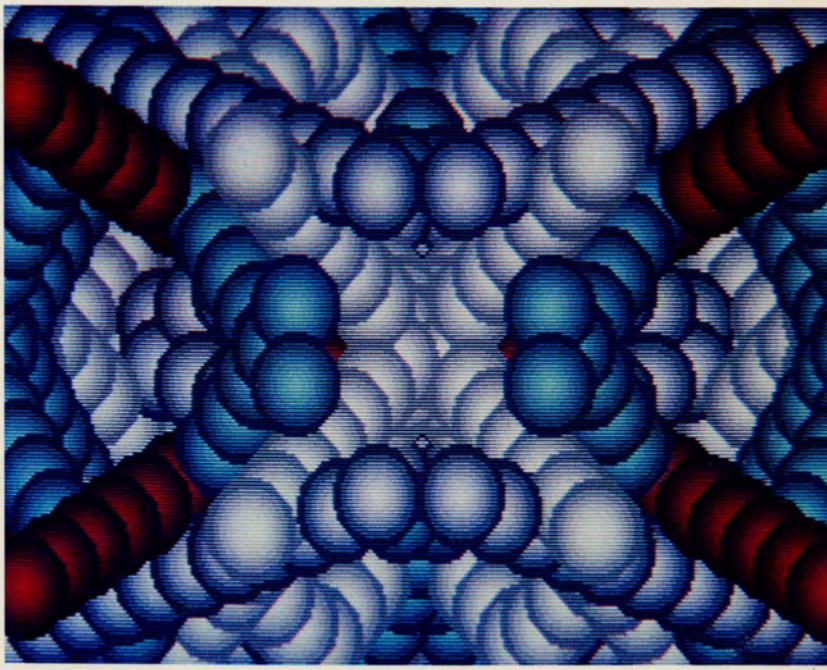
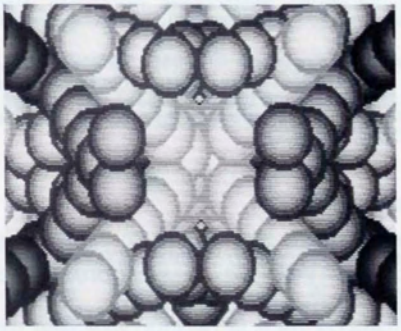
AMIGA™

ABasiC™



AMIGA

ABasiC



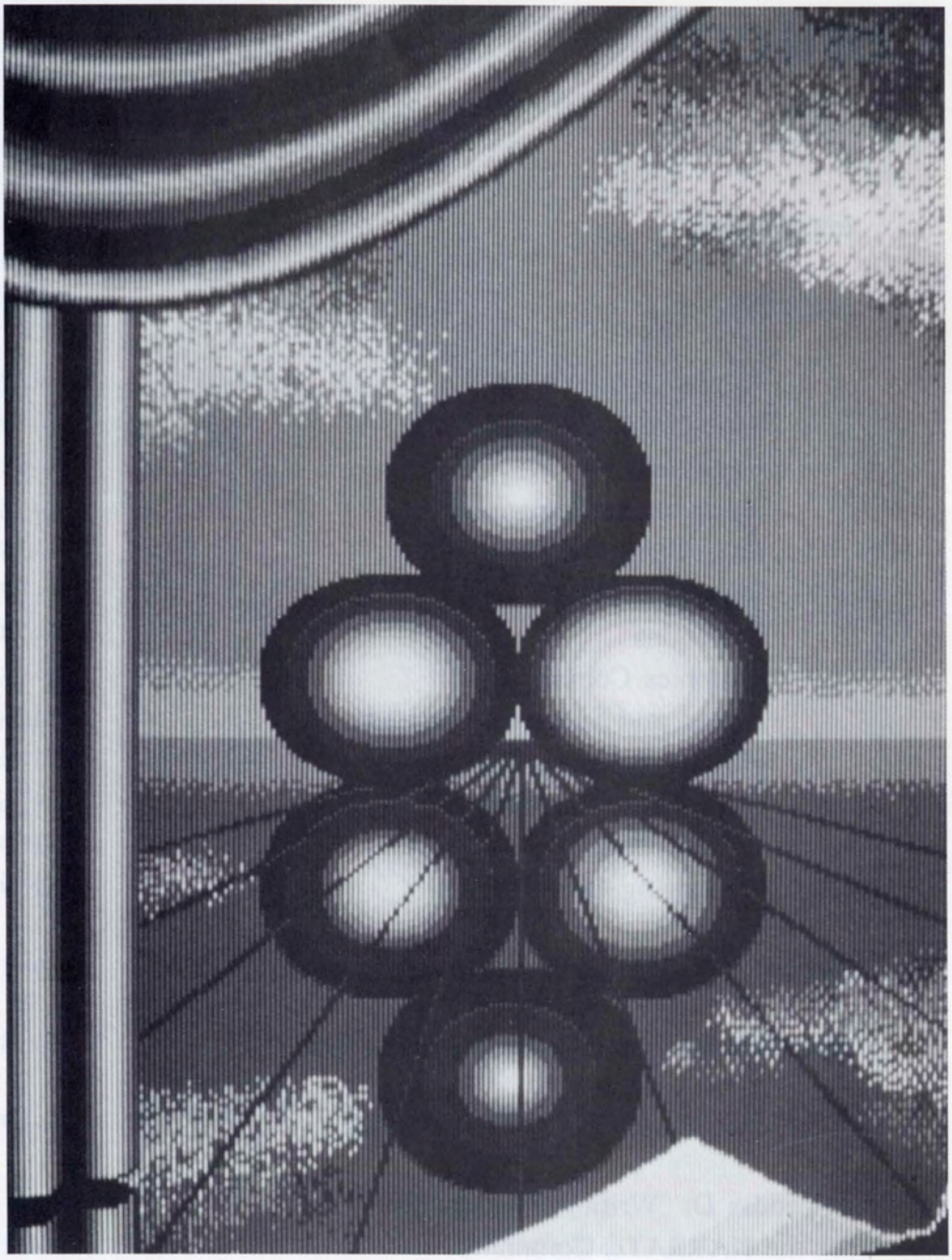
Manual text written by Margaret Steimer

ABasiC language design: Robert Peck, Margaret Steimer, and Metacomco (a division of Tenchstar, Ltd.)

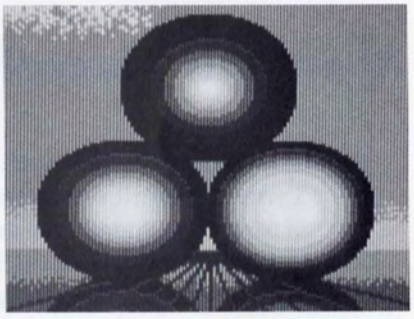
Printed in U.S.A.

CBM Product Number 372102-01 Rev. A

Contents



Introduction



Perhaps the best part about working with computers is that *you* get to control what happens on the screen. Most computer applications and games allow you some degree of control. But you really need a programming language to gain access to all of the computer's power and flexibility. The rewards are endless.

Your Amiga is an extraordinary machine. To harness its many talents, we have developed an extraordinary programming language--ABasiC. ABasiC is like ordinary BASIC in many ways, yet it gives you easy access to the Amiga's high-speed graphics, sound, speech, and much more.

Why BASIC? Because its instructions to the computer are similar to English sentences, BASIC is easy to learn and use. Because BASIC enjoys great popularity, you're more likely to be somewhat familiar with it on some level. This means you can get to the fun part of programming your Amiga right away.

If you're a beginner, you'll want to try the simple operations first. Just remember that no programming mistake you make can hurt your Amiga. So don't be afraid to experiment. If you do have some programming experience, you'll want to explore all of the exciting capabilities this machine offers. ABasiC provides an impressive set of programming tools, including a flexible means of accessing the Amiga's resident library routines and your own machine language routines.

About this Manual

This manual consists of three sections. The first section, "Definition of Terms," describes the terminology and conventions used in the reference section. Included are descriptions of the kinds of variables you can use, their ranges and naming limitations, and the arithmetic operators and their use as part of ABasiC statements.

The "Reference" section fully describes each of the commands available in ABasiC. When you have only a vague idea of which screwdriver you want out of your workroom, you usually know at least which drawer to look in. We have used a similar approach in organizing the ABasiC command descriptions. The commands are logically grouped—Graphics, File Management, Sound and Speech, and so forth—and are alphabetically listed within each category.

Each command description includes its allowable syntax (that is, how you must formulate each instruction so the Amiga can understand it). Also included is a description of any options, extra information the command requires, and so forth. Most command descriptions include a simple programming example.

The last section, "Appendices," provides further reference material for your programming needs, as follows:

Appendix A: Quick Command Reference

Lists each command in the order you'll find it in the Reference section. Following the command name is a brief description of the command and the page number where you can find details. Once you're familiar with ABasiC, you'll be able to do most of your programming by just referring to the Quick Command Reference.

Appendix B: ASCII Character Codes

Lists the characters that correspond to each of the ASCII codes.

Appendix C: ABasiC Error Codes and AmigaDOS error codes

Lists the code number and the associated message for each of the errors that can be generated in ABasiC. Following this is a list of the AmigaDOS error codes.

Appendix D: Writing Phonetically for the NARRATE Command

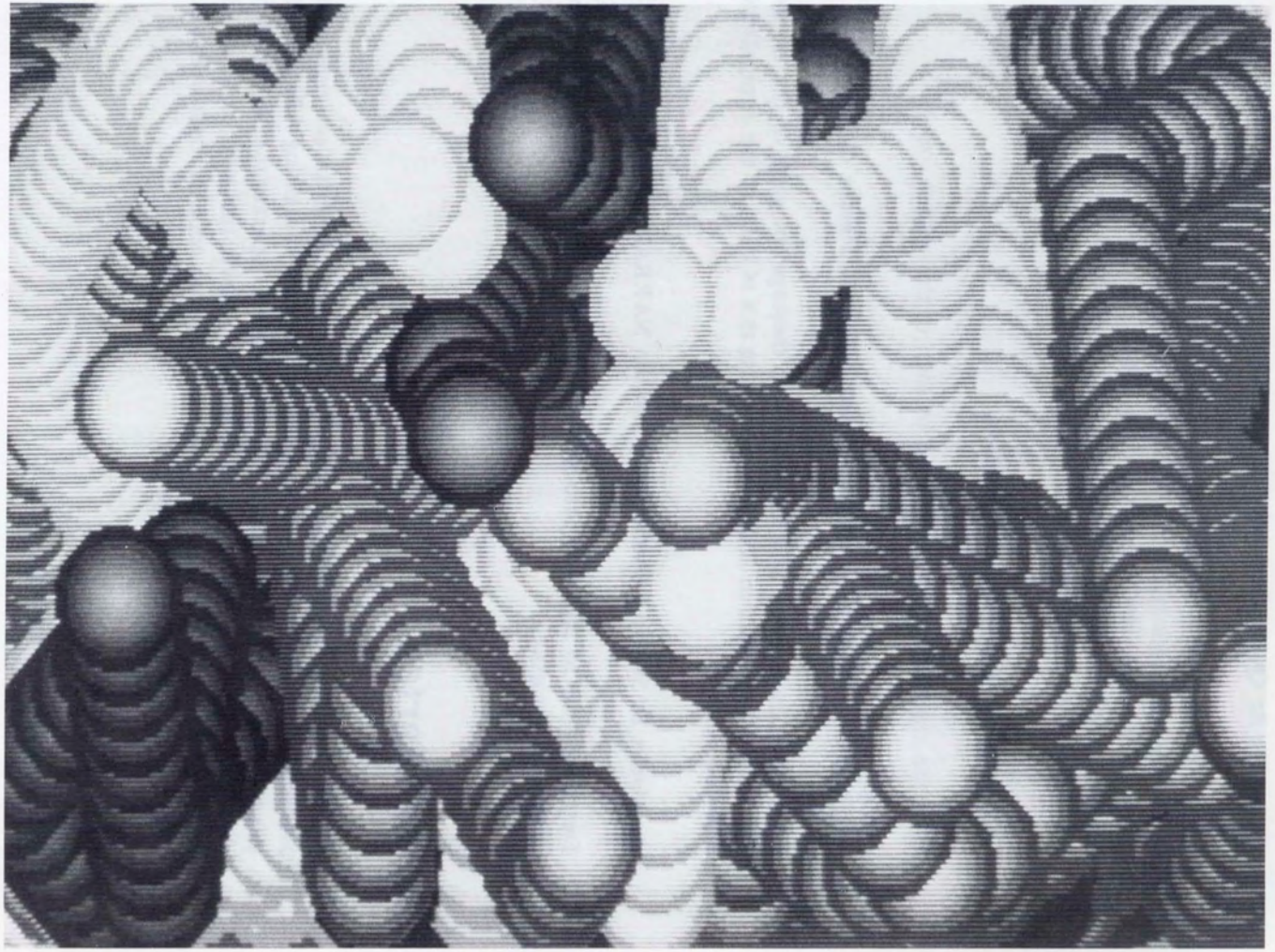
Describes how to turn text into phonetic strings for the Narrator speech synthesizer to use.

Getting Started

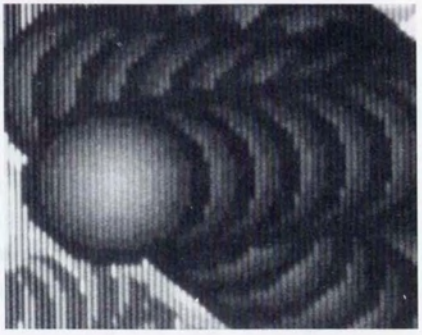
To get started programming in ABasiC, turn on your Amiga and open the *Workbench*. Next, make a working copy of the disk containing ABasiC and put the original in a safe place. To learn how to operate the Workbench and copy disks, see *Introduction to Amiga*. Insert the working copy of ABasiC into the disk drive. When the ABasiC icon appears, open ABasiC. As soon as you see the prompt

```
ok
```

you're ready to program!



ABasiC Reference



This reference provides the information you need to write ABasiC programs--at your fingertips. This information includes a definition of terms and symbols, command descriptions, and other information you need to use the commands. The commands are grouped by function and listed alphabetically within each category. At the end of this manual, there is a quick reference of the categories and commands. Most command descriptions include a programming example in a shaded box that follows the explanation.

Definition of Terms

Elements

ABasiC programs consist of a number of *elements*, items of information that the internal programs need to understand your instructions. You can assign names to some of these elements and control their values; these are called *variables*. With some restrictions, you can assign to a variable any name you wish.

You must enter other elements exactly as they appear in this reference; these elements are *commands* and *keywords*. A *command* is like a verb; it tells ABasiC to perform an action. For example, you tell ABasiC to print something with the PRINT command. You then modify most commands in some manner that tells ABasiC just how to perform the action, what values to use, and so on.

Values that a command needs in order to perform its action are called *arguments*, or *parameters*. Each ABasiC command has its own form and kinds of values; the command descriptions include this information where it applies. Certain commands use one or more *keywords*, which further specify the kind of action to take.

Commands and keywords make up the *reserved words* in ABasiC. You must spell these words correctly. You may know what is intended when you see "PRIMT," but a computer doesn't. On the other hand, ABasiC does allow commands and keywords to appear in either upper or lower case—or a combination of the two. It treats the variable abc the same as ABC or aBC.

Certain symbols and characters are command modifiers called *operators*. Operators combine numbers and characters to form *expressions*. Like an algebraic expression, an ABasiC expression must be stated in a certain way and follow stated rules. These rules follow conventions as nearly as possible—that's one reason ABasiC instructions are easy to read. For example, you use the plus symbol to perform addition and parentheses to group terms.

Statements

A *statement* is a complete instruction for the computer to carry out. This includes a command and--where applicable--keywords, arguments, expressions, and other modifiers. Statements can be quite complex or as simple as a single command. For example,

```
END
```

is a valid and complete statement.

The statement *syntax* is the order in which the statement's command and other modifying elements appear. ABasiC informs you of a syntax error--i.e., a misspelled keyword, an improper sequence of elements, or an illegal form of command modifier--with an error message, "Syntax error."

You must separate each command and keyword in an ABasiC statement, and most of the elements that modify them, with a blank or another acceptable character, called a *delimiter*. Blanks are the most common delimiters, although tab characters, commas, semicolons, and several other characters also separate elements.

Most delimiters, however, have other specific uses. For example, a pound sign (#) following the PRINT command instructs the computer to send subsequent information to a file rather than the monitor screen. A semicolon following an item to be printed prevents ABasiC from issuing an automatic line feed and carriage return. Each special delimiter is described with the appropriate command or operator.

Logical Lines and Program Lines

Several statements can be combined into a single instruction called a *logical line*. (The word "logical" distinguishes such an instruction from a physical line on the monitor screen.) The colon (:) is a special delimiter that separates the statements from one another in a logical line. The only limit to the number of statements you can combine is the allowable number of

characters per logical line. This limit is 255 characters in ABasiC. To end a logical line, press Return.

When you place an integer between 1 and 65529 in front of a logical line, you create a *program line*. If you type a valid ABasiC statement without a line number, the statement is executed the instant you press Return. This is called *immediate mode* execution.

On the other hand, you can enter as many program lines as available memory allows; none of these work until you give the order. This is called *program mode*, or *deferred mode*, since execution is deferred until the desired time. By pressing Return after entering a program line, you simply end the line.

When you run, or *execute*, a program, ABasiC follows the instructions in the order of the line numbers. Normally, the lowest numbered line executes first, followed by successively higher numbered lines. Exceptions to this order of execution are described under "Program Flow Commands." You can enter program lines in any order you wish and retype unsatisfactory lines; ABasiC takes care of keeping them in order.

Constants

Constants are numbers or character strings that don't change during program execution. The different types of constants and rules on their use in ABasiC follow.

Integers

You can use any integer that falls within the range of positive or negative 2,147,483,647. In addition, ABasiC treats a number with floating point characteristics (see below) as an integer if it ends with a percent sign (%), such as 12345.76%. Numeric constants cannot contain a comma. For example,

```
3,500
```

is not a valid constant in ABasiC; it must be represented as

```
3500
```

If you enter a number with a percent sign that falls outside the acceptable integer range, an error results.

Base–10 numeric notation (decimal numbers) is the default (built–in) number base in ABasiC. You can specify integers in other bases as follows:

&B Before a number denotes binary, or base–2, numbers. For example, &B001001 equals 9 decimal.

&O Before a number denotes octal, or base–8, numbers. For example, &O234 equals 564 decimal.

&H Before a number denotes hexadecimal, or base–16, numbers. For example, &H234B equals 9035 decimal. Hexadecimal digits beyond those representing base–10 numerals use the letters A through F:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F 10, etc.
```

&" Before a character denotes a character constant. The ASCII value of the character is stored in the integer. For example, &"A stores ASCII 65, the code for A.

Floating Point Numbers

A *floating point* number is a constant with one or more of the following characteristics:

- It has a decimal point.
- It is expressed in scientific notation.
- It ends with a pound sign (#) or an exclamation point (!).

Floating point numbers can be either single–precision or double–precision, depending on the available memory and the required degree of accuracy. Single–precision numbers are internally accurate to 6.8 digits, although they print showing six digits of accuracy (in other words, up to six digits to the right of the decimal point). Double–precision numbers are accurate both internally and externally to 15 digits.

ABasiC automatically assigns double precision to a number with a fractional part greater than six digits, unless you end the number with the special terminators ! or %. For example,

```
98.3435765!
```

is treated as single–precision, even though it exceeds six digits to the right of the decimal. You can force ABasiC to treat any integer or single–precision number as a double–precision number with the special terminator #.

Scientific notation is used to represent floating point numbers that are either too large or too small for ABasiC to display as simple numbers. The form is as follows:

```
mantissa   e (or d)   exponent
```

where the mantissa (the decimal part) is a positive or negative number, and the exponent is a whole number between −38 and +38 for single–precision numbers (denoted by e), or between −308 and +307 for double–precision numbers (denoted by d). The exponent represents the number of places to

move the decimal point to the right (if positive) or left (if negative) to obtain a simple representation of the number.

You can represent any number in scientific notation. ABasiC only converts numbers to this form automatically if they are larger or smaller than the allowable limit for simple representation.

String Constants

Frequently, you'll want ABasiC to use a series of characters exactly as you enter them; such a series is called a *string constant*. ABasiC recognizes virtually any series of characters in the ABasiC character set that it finds between double quotes (") as a string constant. For example,

```
"FOR X = 1 TO 17"
```

looks like an ABasiC statement, but the double quotes prevent ABasiC from treating it as such. If a string declaration comes at the end of a logical line, the ending double quote is not required. For example:

```
10 IF X = 10 THEN PRINT "THIS IS THE END OF THE LINE
```

Even though you can enter ABasiC reserved words in either upper or lower case, there is a difference between an upper and a lower case character within a string. The ASCII code for "A," for example, is not the same as that for "a."

You can generate a *null* string, one that contains no characters, with two sets of double quotes. For example,

```
A$ = ""
```

assigns an empty character string to A$.

The limit for a character string is 255 characters. If you wish to include a double quote as a *part* of the string constant, type the double quote twice:

```
10 PRINT "They called her ""Motormouth Marge."""
```

produces the output

```
They called her "Motormouth Marge."
```

Variables

Programs typically perform calculations on numbers and character strings that change in value. *Variables* represent items in your program that have different values as conditions change. Normally, you assign a starting value to a variable (that is, you *initialize* it) at the same time you name, or *declare,* it.

When you declare a variable, ABasiC reserves a certain amount of memory for its value. The exact amount of memory depends on the type of variable. Like ABasiC reserved words, variable names can be entered in any combination of upper and lower case letters. There are two kinds of variables--numeric and string. Each of these is described below.

Numeric Variables

ABasiC treats any variable ending in a letter or number as a single-precision floating point number, the default variable type. You must end the names of other variable types--integers, double-precision floating point, and string--with special terminators in order for ABasiC to recognize them as such. Numeric variable type declarations are optional. However, they are useful in some applications. Numeric variable type declarations are listed below:

% Denotes a variable with 32–bit integer values.

! Denotes a single–precision floating point variable. (Because this is the default variable type, the use of this symbol is not necessary.)

# Denotes a double–precision floating point variable.

You can give variables any name you wish, provided you don't use any of the ABasiC reserved words. Names can be up to 255 characters long, although only the first 31 characters are significant. (ABasiC ignores additional characters.) All letters and numerals are valid characters in variable names; punctuation symbols are not valid characters. Naturally, a blank is not a valid character in a variable name, since it is a delimiter.

Some examples of valid variable names are:

```
Integer:            Geronimo%, X%, POINTER1%

Single–precision:   N!, NUMBER, nanosecond

Double–precision:   Nonsense4#, HARRY#
```

When available memory is limited, assign single–precision variables where possible. The determining factor is how large the variable's value might be. With a large program, you can save a surprising amount of memory in this manner.

In case of any conflict between a variable type and its value, the variable type always takes precedence. For example, if ABasiC encounters the statement:

```
I% = 15.56
```

it rounds the value to the nearest integer, 16, because I% is an integer variable type. (Conversely, if you assign a single–precision constant to a

double–precision variable, the data convert to double precision but only to the original single–precision degree of accuracy.)

If there is a conflict between the value generated and the limits on the value for that variable type, an error results.

Arrays

An *array* is a variable that can store an entire list of values in an almost unlimited number of dimensions. There is a complete description of array declaration and its use with the DIM (DIMension) statement described under "Assignment Commands."

String Variables

If you wish to declare a variable that stores a character string, rather than a number, end the variable's name with yet another special character––the dollar sign ($). Actually, you *can* enter a number for the value of a string variable (but not vice versa), but the number is regarded as a string. For example, a telephone number such as

```
(415) 767-1212
```

can *only* be represented as a string, since it contains punctuation. Some examples of valid string variables are:

```
Violet$,  MONEY$,  lollipop$
```

Arithmetic Operators

Arithmetic operators usually combine two terms in a numeric expression. Such a combination is a *binary* operation and results in a single arithmetic term. Arithmetic operators are also delimiters, so you don't have to separate them from the terms with a blank.

Numeric expressions in ABasiC can be constants, variables, or a combination of both, which are joined by other operators, functions, or parentheses. The plus operator is also used in string expressions. The general form of a numeric expression is:

```
<numeric expression>  operator  <numeric expression>
```

The arithmetic operators you can use in creating numeric expressions are described below.

Symbol	Description	Result
+	Adds the two expressions.	Sum
–	Subtracts the expression on the right from the expression on the left.	Difference
*	Multiplies the two expressions. **Note:** The convention of placing factors together to denote multiplication is not valid in ABasiC; you must use an explicit symbol.	Product
/	Divides the expression on the left by the expression on the right. If the value of the expression on the right is zero, an error occurs.	Quotient
\	Converts both expressions to integers and divides the expression on the left by that on the right. The answer truncates to a whole number.	Integer Quotient

Symbol	Description	Result
^ (or **)	Raises the numeric expression on the left (the *operand*) to the power specified by the expression on the right (the *exponent*) and returns the resulting value. Two asterisks (**) are an alternate form of this operator. A negative exponent returns the reciprocal of the operand raised to the power of the exponent's absolute value. For example, 5^(−3) is equivalent to (1/5)^3 or 1/125, expressed as .008 in the computer. A fractional exponent returns the appropriate root of the operand: the expression 81^(1/4) returns the fourth root of 81, which is 3. Note: You can enter an exponent in fractional form (e.g., 1/4); however, ABasiC converts it to decimal form, so it reads 1/4 as .25. Irrational numbers such as 1/3 can produce some roundoff error, so that the result may be close to but not exactly a whole number. (For example, 216^(1/3) doesn't produce exactly 6.)	Exponentiation

```
10 A=1296^(1/4)
20 B=A^2
30 PRINT B
```

When you run the program, the result is:

```
36
```

Special Uses of Arithmetic Operators

+ (as a string operator) `"string" + "string"`

You can use the plus (+) operator to *concatenate* (join together) two string expressions. ABasiC combines the string expressions to make a new string. Mixing string and numeric expressions using the + operator produces an error.

```
10 A$="SHE STUCK TO HIM" + " LIKE GLUE"
20 PRINT A$
```

When you run the program, the result is:

```
SHE STUCK TO HIM LIKE GLUE
```

− (Unary) `- <numeric expression>`

When the − operator precedes a numeric expression, it is a *unary minus*. A unary minus negates the expression; it is the same as subtracting the expression from zero. A unary minus affects only the term that immediately follows it. It's a good practice to separate the term you want to negate from preceding expressions, operators, and functions by enclosing the minus and its expression in parentheses. For example, to multiply 7 by −5, use this expression:

```
7*(-5)
```

The parentheses separate the − and * operators.

```
10 A=32*(-6)
30 PRINT A
```

When you run the program, the result is:

```
-192
```

See "Input/Output Commands" for a description of the use of the + and – symbols in the PRINT USING command.

Grouping Delimiters and Precedence

ABasiC normally evaluates a numeric expression from left to right, using the following order of operations: 1) exponentiation, 2) negation with unary minus, 3) multiplication and division, and 4) addition and subtraction. In general, ABasiC first performs multiplication and division in the expression, from left to right; then it goes to the beginning of the expression and performs addition and subtraction on the results of the first calculations.

The following example shows the exact sequence ABasiC follows in evaluating a complex expression:

```
X*Y+Z*W+P/Q

1) Multiply X*Y; result is T1
2) Multiply Z*W; result is T2
3) Add T1+T2; result is T3
4) Divide P/Q; result is T4
5) Add T4+T5
```

You can bypass this *precedence* by enclosing a portion of the numeric expression in parentheses. The left or right parenthesis serves as a legal delimiter, so an additional blank isn't necessary. ABasiC evaluates all

expressions inside parentheses, following the precedence rules within the inner expression, before doing any other calculations. Where there is one pair of parentheses inside another, ABasiC evaluates the inner set first.

Below are examples of arithmetic expressions and equivalent valid expressions in ABasiC.

Arithmetic Expression	**Equivalent ABasiC Expression**
$3X^2+ 7$	3*X^2+2
$3(X + 2)^2$	3*(X+2)^2
$5-Q(4R^3)$	5-Q*(4*R^3)
$\frac{X}{2Y-X}$	X/(2*Y-X)

Conventions Used in This Reference

The command descriptions in this reference include format examples that follow the conventions described below. The format to the right of a command name shows the order of elements, and required and optional modifying elements for the command. These format conventions are as follows:

Term or Symbol	Description
<numeric expression>	Any number, numeric variable, or valid combination using the arithmetic operators and parentheses. Other expressions are used between these symbols where appropriate.
"string"	Any string constant is allowed; you must include the quotes. You can also substitute any string variable, without quotes, except where noted.
[ ]	The square brackets indicate an optional portion of the statement; they are *not* a part of the statement.
|	Separates mutually exclusive choices in a statement.
...	Indicates that the series can be continued.
<variable list>	Depending on the application: <numeric variable> , <numeric variable> ... or string variable> , <string variable> ...

Line Editor Commands

ABasiC offers a simple means of correcting or extending existing program lines. You can always delete a single line by typing its number followed by Return. The DELETE command (see "Systems Commands") removes a range of lines.

When you type a line with a mistake in it and press Return, ABasiC immediately tells you the line contains an error. If the line was entered in immediate mode, it simply doesn't execute. Instead, the edit arrow points to the problem, as shown below:

```
FOR I = 1 TWO 10
          ↑
```

ABasiC displays the following message:

```
Syntax Error
```

If the incorrect line has a line number--for example, 30--ABasiC tells you the line has a mistake. If you try to run the program, the program halts at that line and displays the following message:

```
Syntax Error at line 30
Program will not run
```

You can then enter edit mode to correct line 30. Below are descriptions of the EDIT command and the special line editing operations that are available in edit mode.

EDIT

EDIT <line number>

Use the EDIT command to enter edit mode. For example, enter the following statement to edit line 120:

```
EDIT 120
```

When you press Return, the following prints:

```
120 REM This is a sic line
```

Below are the operations you use to make changes to the line once you are in edit mode. In general, you enter an edit command and press Return. The line reprints showing the effect of that command. If the command places the line editor in insert mode, all characters you type are inserted where the cursor is positioned in the line. Return or Esc end insert mode.

Cursor Movement

To move the screen cursor to the line you wish to edit, use the arrow keys and edit commands L and R.

Operation	Description
↑	Moves the cursor up one line on the screen (if on the first line, the line reprints).
↓	Moves the cursor down one line on the screen (if on the last line, the line reprints).

Operation	Description
→	Moves the cursor to the right one character (if at the end of the line, moves it to the beginning of the next line).
←	Moves the cursor to the left one character (if at the beginning of the line, the line reprints).
L	Moves cursor left to the beginning of the logical line.
R	Moves cursor right to the end of the logical line.

Character Entry and Deletion

To insert or delete one or more characters in the program line, use the following commands:

Operation	Description
Esc	Exits from insert mode. ABasiC prints a dollar sign ($) and the cursor remains in the same position. (The dollar sign is not part of the resulting line.)
<n> C	Deletes n characters to the right of the cursor and enters insert mode.
<n> D	Deletes n characters to the right of the cursor.
H	Deletes the characters between the cursor and the end of the line and enters insert mode.

Operation	Description
I	Enters insert mode. When insert mode is on, the characters you type go in front of the remainder of the line; existing characters are not overwritten.
<n> K <character>	Searches for the nth occurrence of the given character and places the cursor under it. The cursor deletes all characters between its position and the searched for character. If it doesn't find the character, the line does not change.
<n> S <character>	Searches for the nth occurrence of the given character and places the cursor under it.
X	Extends the line: moves the cursor to the end of the program line and enters insert mode.
Z	Deletes the carriage return at the end of the line.

Additional Commands

The following commands affect the entire line being edited:

Operation	Description
A	Ignores all changes to the line and begins editing again.
E	Ends edit mode and saves all changes to the line.
Q	Ends edit mode and ignores all changes to the line.

Operators

Relational Operators

A relational operator determines whether a specific relationship between two expressions is true. It returns a value of −1 if the relationship is true and a value of 0 if the relationship is false. You don't directly perceive the result of the comparison; rather, you use the result to control your program's execution.

You commonly use relational operators with an IF...THEN command (see "Program Flow Commands" for details). The IF...THEN command uses the −1 or 0 (called a *flag*) to determine what action to take. Relational operators can also stand by themselves without the IF...THEN command; however, you won't often use that form.

A relational operator can test the relationship between either numeric or string expressions, but you can't mix the two types. If you attempt to compare a numeric expression with a string expression, an error results.

Relational operators compare numeric expressions in terms of their values. In the case of string expressions, on the other hand, they compare the ASCII values character by character, from the left to the right of each string. One character is considered less than another if its ASCII code is lower. (If you're dealing only with letters, strings are compared in alphabetical order: A is less than C; J is more than H, etc.) If two strings are equal except for length, the shorter string is considered less than the longer string. Blanks are treated as characters in a string.

Arithmetic operators take precedence over relational operators, which in turn take precedence over logical operators (see below). ABasiC treats relational operators as equal in terms of order of precedence.

Below are descriptions of the relational operators you can use in comparing expressions. The general form is:

```
<expression>  operator  <expression>
```

Symbol	Description
=	Returns a −1 (true) if the two expressions are equal, and a 0 (false) if they are not equal.
<	Returns a −1 (true) if the expression on the left is less than the expression on the right, and a 0 (false) otherwise.
>	Returns a −1 (true) if the expression on the left is greater than the expression on the right, and a 0 (false) otherwise.
<=	Returns a −1 (true) if the expression on the left is less than or equal to the expression on the right, and a 0 (false) otherwise.
>=	Returns a −1 (true) if the expression on the left is greater than or equal to the expression on the right, and a 0 (false) otherwise.
<>	Returns a −1 (true) if the two expressions are not equal, and a 0 (false) if they are equal.

```
10 INPUT "YOUR AGE"; N
20  IF N < 0 THEN PRINT "Who are you kidding?"
30  IF N = 0 THEN PRINT "Welcome to Earth!"
40  IF N >= 5 THEN PRINT "What grade are you in?"
50  IF N <= 69 THEN PRINT "Keep on Truckin'!"
60  IF N > 100 THEN PRINT "Congratulations!"
```

When you run the program, the output varies, depending on the values typed in. Here are a couple of examples:

```
YOUR AGE? -248                  (number input by user)
Who are you kidding?
Keep on truckin'!

YOUR AGE? 113                   (number input by user)
What grade are you in?
Congratulations!
```

Logical Operators

You typically use logical operators inside an IF...THEN command to combine the results of relational operators. (See "Relational Operators" for details.) The comparisons using relational operators return values that can, in turn, be combined to yield a single truth value using logical operators. The result of this combination is either a true flag (−1) or a false flag (0). ABasiC uses −1 to denote truth because it is the logical complement of 0 (that is, NOT (0) = −1 and NOT (−1) = 0.)

(Note: You can also use logical operators to combine binary numbers using Boolean logic; examples are given at the end of this section.)

AND

<relationship> AND <relationship>

AND combines the results of two relationships (or numbers) and declares truth (−1) only if both relationships are true. If one relationship is true and the other is false, or if both relationships are false, AND returns a false flag (0).

```
10 INPUT NAME$
20 INPUT AMOUNT
30 IF NAME$ = "ALFONZ" AND AMOUNT > 10000 THEN PR
   INT "ALFONZ IS RICH!"
```

When you run the program, the result is:

```
?ALFONZ              (entered by user)
?15678               (entered by user)
ALFONZ IS RICH!
```

EQV

<relationship> EQV <relationship>

EQV (EQuiValence) combines the results of two relationships and declares truth only if both relationships are true or both are false. You obtain the same result by combining XOR and NOT. If one relationship is false and the other true, EQV returns a false flag.

```
10 A= -45
20 B= -26
30 IF A>=0 EQV B>=0 THEN PRINT "A AND B FALL IN THE
SAME RANGE"
```

When you run the program, the result is:

```
A AND B FALL IN THE SAME RANGE
```

IMP

<relationship> EQV <relationship>

IMP (IMPlication) is a Boolean logical version of the ABasiC IF...THEN command. If the first relationship is false, IMP returns a true flag regardless of the truth of the second relationship. If the first relationship is true, the value depends on the second relationship: IMP returns a true flag if the second relationship is true and a false flag if the second relationship is false. The truth flags work as follows:

True	IMP	true	returns true
True	IMP	false	returns false
False	IMP	true	returns true
False	IMP	false	returns true

You won't often use IMP with an IF...THEN command. It appears more often as an advanced programming technique to logically combine the contents of two memory locations.

NOT

NOT <relationship>

The logical operator NOT inverts the truth of a single relationship or expression. If the relationship is true, NOT changes it to false, and vice versa. Since it doesn't compare two relationships, you might actually consider NOT more as a function than a logical operator. As the following example shows, your code is clearer if you enclose the inverted expression in parentheses.

```
10 A = 5*6
20 IF NOT (A <= 25) THEN PRINT "LARGER THAN 25"
```

When you run the program, the result is:

```
ONLY ONE STATEMENT IS TRUE
```

OR

<relationship> OR <relationship>

OR combines the results of two relationships and declares truth if one or both of the two relationships is true. If both relationships are false, then OR returns a false flag.

```
10 NAME$ = "CHICO"
20 IF NAME$ = "CHICO" OR NAME$ = "HARPO" OR NAME$
 ="GROUCHO" THEN PRINT "HELLO, MR. MARX."
```

When you run the program, the result is:

```
HELLO, MR. MARX.
```

XOR <relationship> XOR <relationship>

XOR (Exclusive OR) combines the results of two relationships and declares truth only if one relationship is true and the other is false. If both relationships are true, or if both are false, then XOR returns a false flag.

```
10 A= 37
20 B= 80
30 IF A=0 XOR B=80 THEN PRINT "ONLY ONE STATEMENT IS
   TRUE"
```

When you run the program, the result is:

```
ONLY ONE STATEMENT IS TRUE
```

For Advanced Programmers

You can use numeric expressions in place of relationships with Boolean operators. These expressions are combined using Boolean logic. Numeric expressions must be integers in the accepted integer range. Fractional numbers are converted to integers.

ABasiC combines two integers using Boolean logic by treating them as binary numbers. It combines them bit by bit, operating on corresponding bits. Below are the truth tables for each Boolean operator in ABasiC.

AND	OR
1 AND 1 = 1	1 OR 1 = 1
1 AND 0 = 0	1 OR 0 = 1
0 AND 1 = 0	0 OR 1 = 1
0 AND 0 = 0	0 OR 0 = 0

XOR	NOT
1 XOR 1 = 0	NOT 1 = 0
1 XOR 0 = 1	NOT 0 = 1
0 XOR 1 = 1	
0 XOR 0 = 0	

EQV	IMP
1 EQV 1 = 1	1 IMP 1 = 1
1 EQV 0 = 0	1 IMP 0 = 0
0 EQV 1 = 0	0 IMP 1 = 1
0 EQV 0 = 1	0 IMP 0 = 1

The following example shows the Boolean operator AND used with two integers:

```
PRINT 7959 AND 264

results in:

256

This is how ABasiC combined the two integers:

0001111100010111                    (Binary for 7959)
             AND
0000000100001000                    (Binary for 264)
----------------
0000000100000000                    (Binary for 256)
```

Assignment Commands

You normally create and *initialize* variables (that is, assign a beginning value) with a single program statement. You use the commands below to assign the different types of variables available in ABasiC.

=

(as a variable assignment command) `<variable> = <expression>`

You can use the = operator to assign a value to a variable. Traditional BASIC requires the LET command (see the description of LET below), but many dialects of the language—including ABasiC—treat LET as an optional command.

To make a variable assignment with the = operator, the variable name must stand alone to the left of the operator, and the expression (numeric or string) must be to the right. Below are a couple of examples:

```
TEMP = 98.6

WOMBAT$ = "A FURRY ANIMAL"
```

The variable name, equal sign, and expression must stand alone in a statement. If you include these elements within a larger expression or use them with a command other than LET, ABasiC interprets the equal sign as a relational operator. (See "Operators" for details.)

CLR

CLR

Use the CLR command to clear file buffers, strings, and arrays from memory. CLR provides the resident ABasiC program the maximum amount of free RAM in which to run. ABasic leaves the program and its simple variables (not arrays) alone.

CLR closes all open files. Using CLR erases information to open files. It also erases accumulated data in a file buffer waiting to be sent to a disk file. CLR doesn't notify devices such as a line printer that a file has been closed: for example, the disk drive keeps a file open that CLR has in effect closed. To avoid file errors and loss of information in file buffers use the CLOSE command before using CLR. You can use CLR in a program or in immediate mode.

DIM

```
DIM <variable> (<constant> [ , <constant>]...)
 [ ,<variable> (<constant >,<constant>...)]...
```

Use the DIM command to associate a variable name with a list of elements of one or more DIMensions; such a list is called an *array*. An array of two or more dimensions is called a *matrix*. DIM does two things: it tells ABasiC the form of the array values (e.g., rows, or rows and columns), and it sets the maximum number of elements each dimension can hold. Each array must have at least one element per dimension and can contain up to 15 dimensions. Each array begins with element number zero (0). (See OPTION BASE for exceptions.)

You must specify a DIM statement before assigning any element values to an array in your program. Failure to do so generates an error. An array with no more than 10 elements in each dimension is an exception, because ABasiC automatically allocates enough memory for such an array. The default number of dimensions is four for integers and three for all other variable types. The memory requirements for each type of element are as follows:

Integer	4 bytes
Single–precision	4 bytes
Double–precision	8 bytes
String	12 bytes

If the program encounters more than one DIM statement (or the same one twice) for a single variable name, an error results. An exception is the use of the ERASE command (described below) to release the memory assigned to an array. It's a good idea to place DIM statements near the beginning of your program.

You can use a single DIM statement to assign several arrays. Just separate the items with commas. The following example assigns two arrays, NUMS and ABC$:

```
DIM NUMS(2,20), ABC$(15)
```

You can determine the total number of an array's elements by multiplying the number of elements per dimension. The following DIM statement:

```
DIM A(30,40)
```

sets aside memory for 30 x 40, or 1200 elements.

When you assign values to the array's elements, use the same structure for the array that you used when you declared the array's dimensions. A typical assignment of an element of the array in the preceding example is:

```
A(K,2) = K*3
```

This statement assigns the value K*3 to row K, column 2 of array A( ).

You can also use DIM to assign space for string arrays. The maximum number of characters that ABasiC reserves for each element in the string array is 255, the limit on a string's size.

```
10 DIM X(15), A%(4,4)
20 FOR I = 1 TO 4: X(I) = I*10: A%(1,I) = I*2
30 NEXT I
40 FOR J = 1 TO 4:PRINT X(J), A%(1,J)
50 NEXT J
```

When you run the program, the result is:

```
10      2
20      4
30      6
40      8
```

ERASE

ERASE <array name> [,<array name>]...

Use the ERASE command to redimension arrays (that is, assign a new number of elements per dimension) during program execution. The memory requirements of a particular array might decrease considerably in the middle of the program. Using ERASE, you can recover the memory that the array no longer needs. Then, use a new DIM statement to reserve the array's new memory requirement. If you attempt to use a new DIM statement before first ERASEing that array, you'll generate an error.

```
10 DIM PAYROLL$(20)
20 FOR I = 1 TO 20
30 INPUT NAME$: NAME$
 PAYROLL$(I)=
...
200 ERASE PAYROLL$: DIM PAYROLL$(35,10)
...
310 PAYROLL$(K,2) = "YEAR-TO-DATE"
```

As the example shows, you can redimension the array to include multiple items per employee, as well as additional employees.

LET

[LET] <variable> = <numeric expression> | "string"

The LET command is optional. Early versions of BASIC required its use in assigning variable values, but ABasiC (and most other dialects) now allow you to use the variable name alone on the left side of the statement. See the description of the use of the = operator for assigning variable values. The following two commands are identical in effect.

```
LET X = X - 35

X = X - 35
```

OPTION BASE

OPTION BASE 1 | 0

The OPTION BASE command sets the base for array dimensions. The default is 0, which means the first element in each dimension of an array is numbered 0. The following statement in your program:

```
OPTION BASE 1
```

tells ABasiC to number all array elements starting with 1.

RANDOMIZE

RANDOMIZE <numeric expression>

Use RANDOMIZE to reseed the random number generator. In other words, this command generates a new sequence of random numbers. If you insert calls to the RANDOMIZE command in your program, the numbers that the RND function returns are more varied and thus are more truly random than if you use the RND function alone.

For example, you might use −1 as an argument:

```
RANDOMIZE -1
```

Each time this statement executes, a different value becomes the new random number *seed* (the value used to start a new sequence), thus assuring a maximum variety of RND values (see "Functions").

REPLACE$

```
REPLACE$( <string variable>, <integer-1>
[ ,<integer-2>]) = "string"
```

Use the REPLACE$ command to substitute one character string for another within the value of an existing string variable. The integer expression <integer–1> is the position of the first character of the string you want to replace. The second integer, which is optional, is the number of characters in the replacement string to use. If you omit the second integer, ABasiC uses all the characters in the replacement string up to the number of characters in it or in the original string, whichever is shorter.

```
60 MONTH$ = "SEPTEMBER"
70 D$ = "9AUG85": PRINT D$
80 REPLACE$(D$,2,3) = MONTH$
90 PRINT D$
```

When you run the program, the result is:

```
9AUG85
9SEP85
```

In the example above, the first three characters of MONTH$ replace three characters (beginning at character number two) of D$.

REM

```
REM <character string>
```

Use the REM command to insert REMarks, or comments, in a program. These comments use some memory, though they can provide helpful documentation.

ABasiC ignores any string of characters--including program statements--that follows a REM statement in a program line. Thus, if you add a REMark to a program line that contains executable statements, REM must be that line's last statement.

You'll find that a temporary insertion of a REM command can help debug your code. You can mask off (hide) statements by inserting a REM in front of them and editing the REM out later. The following illustrates the use of REM in program lines:

```
20 REM This program calculates the number of angels that can dance
   on the head of a pin

50 FOR K = 1 TO 17: REM Loop thru # of formulas
```

The apostrophe (') is an alternative form of the REM command. For example, the following ABasiC line is valid:

```
10 FOR K = 1 TO 10: ' Loop thru K lists
```

SWAP

SWAP <variable-1>, <variable-2>

Use the SWAP command to exchange the values of two variables. The variables must be the same data type, such as both single-precision or both string variables. You can SWAP array variables, but not the array values themselves.

```
10 X = 5: Y = 10
20 SWAP X,Y: PRINT X,Y

When you run the program, the result is:

10          5
```

Input/Output Commands

This section describes most of the commands that provide communication between you and the computer. *Input* is information that you give the computer; *output* is information that you receive from the computer. The forms in which you send and receive information vary considerably. The keyboard is the most common source of input, while most output appears on the monitor screen.

Under "File Management Commands," you'll find specific information about input and output commands that relate to data files. That subsection also contains information on routing program input/output to and from other devices, such as the printer. The subsection "Graphics Commands" treats graphics–related input/output, such as control of the mouse.

DATA

```
DATA  <constant>  [, <constant> ]...
```

A DATA statement provides information that your program can use by means of a READ statement; the two commands work together. Use the DATA command to enter and use large amounts of data that have a similar format. A list of constants separated by commas follows the DATA command.

Each item in the list occupies a *field* (a relative position in the list). If the READ statement that the program uses to get information expects four items, the DATA statement must contain four fields of information. A DATA statement that contains fewer fields produces an error. A statement that has too many fields throws off the field count of the corresponding READ statement.

You can use a single READ statement with several DATA statements. For example, you can loop through four DATA statements, each of which has four fields, with a single READ statement, as shown below:

```
10 FOR I=1 TO 4: READ A,B,C$,D: NEXT I
...

100 DATA 5,6,"HELLO",7
110 DATA 1,2,"GOODBYE",3
120 DATA 9,10,"WHY",11
130 DATA 4,5,"BECAUSE",6
```

Suppose the READ statement in line 10 had only three variables: A, B, and C$. It would only read the first three items of the first DATA statement (line 100) on the first pass through the loop. The second time through, READ would get the remaining item in line 100, then the first two items in line 200. The third time through the loop, an error would occur as READ attempted to assign the string constant "GOODBYE" to numeric variable A.

You can mix data types any way you wish, as long as their order is the same as the order of variable types in the matching READ statement. DATA statements can appear in any line in your program without interrupting program flow. Each DATA statement must stand alone on a program line. See the READ command for a programming example using DATA and READ.

GET

```
GET <string variable> [ ,<numeric expression>]
```

Use GET to receive input from the keyboard one character at a time. The input character is assigned to the specified variable. If GET detects no input, the variable's value is the null (empty) character.

The optional numeric expression is an integer constant or variable that indicates the number of microseconds you want the program to wait for keyboard input. If you omit this option, ABasiC interprets it as zero, an instant poll (or input check) of the keyboard.

The character that GET obtains doesn't automatically print on the screen; you must provide a statement to print the variable's value. You can use a GET statement in a loop to force the computer to wait for a keystroke

before it continues program execution. You don't have to press Return after you type a character. In fact, GET registers Return as a character, just as it would any other character.

Suppose you want your program to print text so that other people have time to read it regardless of their reading speed. Your program should print the text followed by a *prompt* (a short message that tells users what input the program expects) to press Return to continue. Then your program should include a loop such as the following:

```
200  GET A$: IF A$  <>  CHR$(13) THEN 200
```

This loop terminates only when the program user presses the Return key (ASCII character 13).

```
30 PRINT  "I never get tired!"
40 GET A$, 10^6 : IF A$ = "" THEN GOTO 30
50 PRINT "Thanks, I needed that!"
```

When you run the program, the result is:

```
I never get tired!
I never get tired!
I never get tired!          (and so on, until a key is pressed)

Thanks, I needed that!
```

GETKEY

GETKEY <string variable>

The GETKEY command, unlike GET, makes the program wait indefinitely for an input character before going on. You don't need to specify a waiting period.

```
120 PRINT = "Waiting for a key..."
...
130 GETKEY A$
140 PRINT "Thanks!"
```

GRAPHIC

GRAPHIC(<integer>)

Use the GRAPHIC command to control the way ABasiC interprets coordinates for printing text on a graphics screen. The numeric expression in parentheses is an integer. A nonzero integer instructs ABasiC to interpret coordinates as pixels. A value of zero, the default, causes ABasiC to interpret coordinates as character coordinates. (In 40–column mode with the default 320 by 200 resolution, each character on the screen consists of a rectangle that is 8 by 8 pixels.) A single GRAPHIC command remains in effect until you either close the window, end the program, or execute another GRAPHIC command that changes the mode of coordinate interpretation.

When you open a custom window (see WINDOW), the program automatically interprets screen coordinates to describe a character location. The GRAPHIC command allows you to change that interpretation from character to pixel coordinates, or vice versa, while the window remains open.

Under default conditions, the PRINT AT command specifies character coordinates for printing text. For example, the following statement prints "HELLO" at character 12, row 10 of the current output window:

```
PRINT AT (12,10); "HELLO"
```

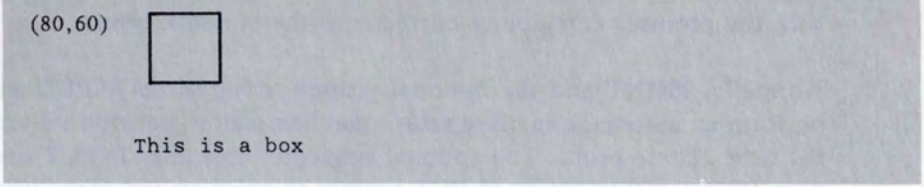
If you precede this same PRINT AT statement with GRAPHIC(n) where n is nonzero, the program interprets the coordinates (12,10) as a pixel location. The result is that "HELLO" prints close to the upper lefthand corner of the output window.

When the output window is in "pixel mode" (that is, you issued a GRAPHIC command with a nonzero argument) you can also position text using the LOCATE command. (See the programming example below.)

GRAPHIC has no effect if your output goes to a device other than the monitor screen.

```
10 GRAPHIC(1)
20 X=80: Y=60
30 BOX(X,Y; X+40,Y+40)
40 LOCATE (X,Y+50)
50 PRINT "This is a box"
60 PRINT AT (X-2-7*8,Y); "(";X;",";Y;")"
```

When you run the program, the result is:



INPUT

```
INPUT [ ; ] [ "prompt string"; | ,]
<variable> [ ,<variable> ]...
```

Use the INPUT command when you want the program to wait for a keyboard response from the user. Unlike GET and GETKEY, INPUT can receive a value that is several characters long. The input for that variable terminates with the Return key. ABasiC stores the response as the variable's value. When the program reaches the INPUT statement, it prints a question mark:

```
?
```

and halts until the user responds.

A question mark doesn't really explain what input the program expects, however. Optionally, you can print a *prompt* (an explanatory message) that tells what kind of response is expected. ABasiC prints a question mark after the prompt if a semicolon follows the print string. If you use a comma after the prompt string, no question mark or blank prints between the prompt and the input response.

You can specify several variables in one INPUT statement. In fact, you can mix different types of variable, if you wish. The input for each variable except the last terminates with a comma. If the person using your program makes an input mistake, the responses must be entered again, starting with the first variable. For example, the following INPUT statement:

```
INPUT A, B, NAME$
```

requires a response of two numbers and a string, separated by commas. Be sure the prompts correspond correctly to the variable types.

Normally, PRINT and the optional prompt string of an INPUT statement perform an automatic carriage return and line feed to position the cursor for the next item to print. The optional semicolon just after INPUT causes the next PRINT statement or INPUT prompt to print on the same line.

```
10 INPUT "NAME, AMOUNT"; NAME$, AMNT

When execution reaches line 10, the user sees the following:

NAME, AMOUNT? LISA,123.45        (The comma separates multiple entries.)
```

If the input variable is numeric and you input a string, ABasiC returns an error. On the other hand, if the INPUT statement expects string input, ABasiC accepts numbers—characters are characters. However, the program recognizes that entry as a string and not as a number.

INPUT#

```
INPUT# <file number>, <variable> [,<variable>]...
```

The INPUT# command is similar to the INPUT command, except INPUT# takes data from a previously opened file instead of the keyboard. The file number must match the number you assigned the file when you opened it. You can't use a prompt string with this form of the command. For details on using INPUT#, see "File Management Commands."

Use the LINE INPUT command to request an entire line of input from the program user. LINE INPUT accepts all characters, including spaces, commas, and quotation marks, as part of the input until Return is pressed. This enables you to get around the restrictions imposed by other input methods, such as not accepting the comma that separates values in the INPUT statement.

Unlike the INPUT command, LINE INPUT doesn't automatically print a question mark after the semicolon following a prompt.

LINE INPUT

```
LINE INPUT [ ; ] [ "prompt string"; | , ]
<string variable>
```

Use the LINE INPUT command to request an entire line of input from the program user. LINE INPUT acccepts all characters, including spaces, commas, and quotation marks as part of the input until Return is pressed. This means that you can get around the restrictions imposed by other input methods such as the comma that separates values in INPUT, if your needs require. Unlike the INPUT command, LINE INPUT doesn't automatically print a question mark after the semicolon following a prompt.

LINE INPUT#

```
LINE INPUT# <file number>, <string variable>
```

The LINE INPUT# command is similar to LINE INPUT, except it takes data from a previously opened disk file instead of the keyboard. The file number must match the number you assigned the file when you opened it. This form of the command does not allow a prompt string. See "File Management Commands" for more details on LINE INPUT#.

PRINT

```
PRINT  [<variable> | "string"] [ , | ; ]
       [<variable-list> |"string"]
```

You can use the PRINT command to display most of your program's text output. There are special forms of PRINT statement for just about any format you wish to appear on the monitor screen. Use the PRINT command for normal text printing. See below for special forms of PRINT using the keywords:

AT()

INVERSE()

USING

You can mix print items such as variable values and strings in any order. You can easily experiment with the appearance of the output, since PRINT works well in immediate mode. Below are a few examples of how to use PRINT. Following these are descriptions of keywords that create special variations on the PRINT command.

Without commas or semicolons, PRINT produces exactly what appears in quotes, or the values of variables, and issues an automatic carriage return and line feed.

```
A$ = "WHAT YOU GET.": PRINT "WHAT YOU SEE IS ": PRINT A$
```

These statements produce the following:

```
WHAT YOU SEE IS
WHAT YOU GET.
```

The colon between PRINT statements forces a carriage return and line feed. If an output string is too long to print on a single line, ABasiC issues an automatic return, which breaks a word in the middle if necessary. You

can print the entire sentence in the above example on a single line, using a semicolon:

```
A$ = "WHAT YOU GET.": PRINT "WHAT YOU SEE IS"; A$
```

These statements produce the following:

```
WHAT YOU SEE ISWHAT YOU GET.
```

You must include a blank after the word "IS" if you don't want the words jammed together, as in this example. Blanks are treated just as any other character within a string, and the semicolon produces no blanks between separate print strings.

The comma inserts tabs between multiple output items as in the following example:

```
X = 2: Y = 3: PRINT X,Y, X*Y
```

These statements produce the following:

```
2           3           6
```

As the preceding example shows, ABasiC calculates the value of numeric expressions in PRINT statements and prints the result.

The Return key ends a logical program line; thus, you clearly can't instruct ABasiC to print a carriage return by simply pressing Return. To force a line feed and carriage return (ASCII 10) before the end of a line, use a statement such as follows:

```
PRINT "PLEASE DON'T FEED" ; CHR$(10) ; "THE LINES."
```

These statements produce the following:

```
PLEASE DON'T FEED
THE LINES.
```

See the CHR$ string function description for details on printing characters (like Return) that are hard to specify directly from the keyboard.

A shorthand form of the PRINT command is a single question mark. For example these two program statements produce identical output:

```
PRINT A,B
? A,B
```

PRINT AT ( )

```
PRINT AT (<x>,<y>);  [<variable> | "string"]
[ , | ; ][<variable-list>| "string"]
```

The PRINT AT command works just like the PRINT command, but requires two values in parentheses at the beginning of the statement to position the print string on the screen. You describe the screen position where you want to start printing in terms of the column (x) and the row (y). You can repeat the AT keyword within the same print string if you want. For example, the following line is valid:

```
PRINT AT (2,Y); "Frag"; AT (12,Y); "men"; AT (24,Y);
"ted"
```

You can also mix AT and INVERSE keywords (see the description that follows) with the same print string.

ABasiC normally interprets the coordinate values as a character position. Alternatively, you can instruct your program to interpret these same values as the coordinates of a pixel position. See GRAPHIC for details.

```
20 NAME$ = "Adam"
30 PRINT AT (5,11); "Madam, I'm " ;NAME$
```

When you run the program, the following prints at character 5 on the 11th row of the screen:

```
Madam, I'm Adam
```

PRINT INVERSE( )

PRINT INVERSE(<integer>) [<variable> | "string"] [, | ;] [<variable-list> | "string"]...

The PRINT INVERSE( ) variation of PRINT lets you print items in *inverse video* on the screen. This means all normal 1–bits become 0–bits on the display, and vice versa. The integer in parenthesis is either 1 or 0. The screen default is INVERSE(0), or white text on a dark blue background; INVERSE causes blue text to print on a white background. The effect of INVERSE (1) lasts to the end of the PRINT statement or until an INVERSE(0) within the same statement.

```
10 FOR K = 1 TO 3
20 PRINT "DARK ";INVERSE (1) "HORSE #"; INVERSE (0) K
30 NEXT K
```

When you run the program, the result is:

```
DARK HORSE #1
DARK HORSE #2
DARK HORSE #3
```

The INVERSE( ) keyword has no effect if program output goes to any device other than the monitor screen.

PRINT USING

PRINT USING "format-string"; <variable-list>

Use the PRINT USING form of the PRINT command to format your output for structures such as tables and graphs. You can align the decimal point in columns of numbers or specify a set number of spaces between strings to be printed.

A format string enclosed in quotes follows the USING keyword. This string defines an "image" of how you want the items in the variable list to appear in the field (that is, the affected screen area). The variable list, which is separated from the rest of the statement by a semicolon, lists the values to be printed. This list can consist of any combination of variable names and literals (actual values).

Below is a programming example with PRINT USING, followed by a description of allowable format string characters.

```
10 X = 32: Y = 1030.23
20 PRINT USING  "$$##.##";13.25,X,Y

When you run the program, the result is:

$13.25          $32.00          %$1030.23
```

The percent sign in front of the last item means that Y's value doesn't conform to the given print format (since it contained six digits). The format uses one pound sign (#) per digit or character to be printed. The double dollar sign causes a dollar sign to print just to the left of the number and reserves one extra character space for the output.

The following discussion lists and illustrates each of the symbols of the PRINT USING command for numeric and string output.

Numeric Output

(pound sign)

The pound sign (#) reserves room for a single character in the numeric output field. If the output has fewer digits than the number reserved, the output is right–justified in the field. More output characters than the number reserved generates a percent sign in front of the number. If no decimal point appears in the format string, any fraction is rounded off.

Format String	Variable List	Result
"##"	12.04	12
	123	%123
	0.8934	1
	1	1

\+ (plus sign)

The plus sign specifies printing of a number's sign. If the symbol precedes the pound signs that reserve digit spaces, the sign prints in front of the number; if it follows, the sign prints after the number.

Format String	Variable List	Result
"+##.#"	4.3	+4.3
	−3.6	−3.6

– (minus sign)

The minus sign generates a trailing "–" sign after a negative number.

Format String	Number	Result
"–##.##"	2.54	2.54
	–2.54	2.54–

. (decimal point)

The decimal point designates the position at which the decimal point should appear in the output. Only one decimal point is allowed; if you don't specify one, output is rounded to the nearest integer. The number of output digits to the left of the decimal point must not exceed the number of corresponding pound signs in the format string. Trailing (that is, following the significant digits to the right of the decimal point) zeroes are printed to fill out the format field.

Format String	Number	Result
"###.##"	–.3	–0.30
	–12.24	–12.24
	12.02	12.02
	–123.34	%–123.34
"##."	5.789	6
	–2.3	–2.

, (comma)

A comma that precedes the decimal point in the format string indicates the placement of a comma before every third digit to the left of the decimal

point in numeric output. The comma also reserves one character in the output field. Thus, if a number requires two commas, you must include sufficient pound signs to reserve enough space.

Format String	Number	Result
"#####,.#"	19454	19,454.0
	4266.73	4,266.7

****** (double asterisk)

The double asterisk pads, or fills in, leading spaces in a numeric field with asterisks. It also reserves two character positions.

Format String	Number	Result
"**####.#"	19454	*19454.0
	6.73	*****6.7

$$ (double dollar sign)

The double dollar sign indicates that a dollar sign should print to the immediate left of numeric output. It also reserves two character positions, one for the dollar sign itself. A minus sign can only appear in a trailing position when you use the double dollar sign.

Format String	Number	Result
"$$#.##"	23.5	$23.50
	.35	$0.35
"$$.##-"	-1.3	$1.30-

**$ (two asterisks followed by a dollar sign)

Two asterisks followed by a dollar sign pad leading spaces in the output field with asterisks. A dollar sign prints to the immediate left of the number. It also reserves two character positions, one for the dollar sign itself.

Format String	Number	Result
"**$.##"	3.5	*$3.50
	.35	*$0.35

^ (caret)

One or more carets indicate that numeric output should appear in scientific notation. At least one pound sign must be used before or after each of these symbols. The position of the decimal point determines the exponent's value; significant digits are left-justified. One character position is reserved to the left of the printed number for the number's sign, unless you explicitly format the sign. (A blank prints for positive numbers.)

Format String	Number	Result
"##.##^"	-100.00	-1.00E+02
	.0036	3.60E-03

_ (underline)

An underline character in a format string indicates the character following it should appear as a literal in the output; that is, ABasiC prints the character itself.

Format String	Number	Result
"#.##_#"	3.5	3.50#
	.35	0.35#

(blank)

A blank following the other characters in a format string separates output fields. (*Any* character that is not part of a format specifier separates output fields.)

Format String	Number	Result
"#.## "	3.5, 4.22, and 9.06	3.50 4.22 9.06

String Output

! (exclamation point)

An exclamation point indicates that only the first character in a string should print.

Format String	String	Result
"! "	Larry Curly Moe	L C M

\..\ (backslashes)

Two backslashes indicate that only two characters from each string using this format are to print. If you add one or more blanks or other characters between the backslashes, ABasiC prints that number of additional string characters. If the field width is greater than the number of characters in the string, the output is left–justified and the remainder of the field is padded with blanks. A period or a digit between the backslashes lets you easily count the total number of characters that will print.

Format String	String	Result
\1234\	Jabberwocky	Jabber
	It	It

& (ampersand)

An ampersand indicates a variable–length string field. Each string using this format prints exactly as entered, regardless of length.

Format String	String	Result
"&"	ABCDEFG	ABCDEFG

READ

```
READ <variable> , <variable> ...
```

Use the READ command to get values from a DATA statement. READ stores these values in variables for program use. The variable list following the READ command can be any combination of string and numeric variables. However, the order and type of data items must correspond to the variable list. If the program attempts to read a string constant into a numeric variable an error results.

You can use a single READ statement with several DATA statements. You can also reuse any DATA statements by resetting the pointer to the desired line with the RESTORE command (described below).

```
10 FOR I = 1 TO 3: READ NAME$,AGE,SPORT$
20 PRINT  "NAME: ";NAME$,"AGE: ";AGE
30 PRINT "FAVORITE SPORT: ";SPORT$
40 NEXT I
100 DATA "MARGE",45,"SOFTBALL"
110 DATA "MURRAY",13,"SOCCER"
120 DATA "MILLIE",89,"BILLIARDS"
```

When you run the program, the result is:

```
NAME: MARGE         AGE: 45
FAVORITE SPORT: SOFTBALL
NAME: MURRAY         AGE: 13
FAVORITE SPORT: SOCCER
NAME: MILLIE         AGE: 89
FAVORITE SPORT: BILLIARDS
```

RESTORE

RESTORE [<line number>]

Use the RESTORE command to point the program back to the beginning of a set of DATA statements. When a program begins executing, READ automatically reads data from the first DATA statement in the program and works its way down by line number. Used alone, RESTORE sets the pointer back to the first DATA statement. If you specify the optional line number, the program restores the data pointer back to that line number.

Suppose one part of your program READs a series of three DATA statements starting at line 100:

```
100 DATA 1,321,456
110 DATA 2,123,789
120 DATA 3,543,987
```

The following statement:

```
RESTORE 110
```

resets the data pointer to the second DATA statement. A different program task can then reuse the last two DATA statements.

SCNCLR

SCNCLR

The SCNCLR command simply clears the screen or the current output window--regardless of whether it shows graphics or text. You should include a SCNCLR command before printing text or graphics in an existing window (or the default window, which is the entire screen).

WIDTH

WIDTH <integer>

The WIDTH command changes the number of characters per line that print on the monitor screen or the line printer. The default is 80 characters. If you want to print more or fewer characters per line (before ABasiC issues an automatic carriage return), use this command. For example, the following statement sets line WIDTH to 20:

```
WIDTH 20
```

The numeric expression should have an integer value.

Screen Position Functions

POS

POS(<integer>)

The POS function returns the number of characters that have printed since the last line feed was issued. The integer result is a value from 1 to 80 on an 80–column screen. If you're using a 40–column screen, any value from 41 to 80 refers to the next line.

Specify the file number of the output window for which you want the position information. The default window has file number 0 (zero), so you normally specify 0 as the argument in parentheses.

```
10 PRINT: PRINT "123"
20 A% = POS(0): PRINT A%
```

When you run the program, the result is:

```
123
3
```

SPC

SPC(<integer>)

Use the SPC function to imbed blanks in a PRINT statement. You can only use SPC within a PRINT statement. The value of the numeric expression in parentheses is the number of blanks to print. Unlike the TAB function, SPC works in relation to the current horizontal cursor position. TAB works in relation to the beginning of the line.

```
30 MONTH$ = "JANUARY": SALES = 345.50
50 PRINT "MONTH"; SPC(8);"SALES"
60 PRINT:PRINT MONTH$; SPC(12-LEN(MONTH$));SALES
```

When you run the program, the result is:

```
MONTH          SALES

JANUARY        345.50
```

Note: The extra PRINT statement at the beginning of line 60 creates an extra vertical space in your output.

TAB

TAB(<integer>)

To insert TABs into your printed output, use the TAB function inside a PRINT statement. The value of the numeric expression is the number of blanks from the beginning of the current line to the TAB. Use this function with PRINT# to insert tabs in a disk file.

```
90 AMT = 123
100 PRINT  TAB(3); "Profits";TAB(2); AMT
```

When you run the program, the result is:

```
   Profits
   123
```

Program Flow Commands

Line numbers inform an ABasiC program of the sequence in which to process instructions. Normal execution steps through line numbers, from the smallest to the largest. There are a number of reasons to interrupt or divert sequential processing though, depending on the circumstances. The following commands describe how you can do so.

Some of the program flow commands require that you specify a line number. The line number can be any integer from 0 to 65529. If you use a fractional number, it truncates. Numbers out of range or references to lines that aren't in the program produce an error.

END

END

The END command halts execution of the current program and returns the computer to immediate mode. END differs from STOP in that END does not return a break message. When ABasiC encounters END, it stops the program and prints:

```
OK
```

END closes all files that are open. The values of variables remain as they are.

Early versions of BASIC required END as the last statement of a program so the computer would know where to stop. In ABasiC, the computer automatically stops at the last statement.

However, END is useful for separating subroutines (see GOSUB) at the end of a program from the main body. Without the END statement, the first subroutine executes without a GOSUB command, so the RETURN command at the end of the subroutine triggers an error.

```
10 PRICE = 5.03
20 GOSUB 100
30 PRICE = 7.30
40 GOSUB 100
50 END
100 TAX = PRICE * .06
110 PRINT "TAX IS: "; TAX
120 RETURN
```

When you run the program, the result is:

```
TAX IS: .3018
TAX IS: .438
```

Without the END statement, the result is:

```
TAX IS: .3018
TAX IS: .438
TAX IS: .438
RETURN without GOSUB at line 120
```

FOR...TO...[STEP]

```
FOR <counter variable> = <start> TO
<limit> [STEP <increment>]
```

The FOR...TO command marks the beginning of a program *loop*. A loop is a series of program instructions that is repeated several times. The FOR...TO command is always paired with a NEXT command, which marks the end of the loop. FOR...TO and its optional extension, STEP, are used to set up a counter variable, starting and ending numbers for the count, and a step increment. The NEXT command, which is the last statement in the loop, uses the variable and values to keep track of the loop repetitions.

When the computer executes the FOR...TO command, it sets the counter variable to the starting value. When program execution reaches the NEXT command, it increments the counter variable by the STEP value. Then it compares the counter to the limit value.

Note: Unless you specify an integer variable as a loop counter, ABasiC treats the counter variable as single-precision floating point. You should

specifically use a floating point variable when you specify fractional STEP values.

If the counter hasn't gone past the limit value, program execution jumps back to the command immediately following the last FOR...TO command and the loop statements repeat execution. If the counter has gone past the limit value, the loop ends and normal program execution continues at the statement following NEXT.

When the program encounters FOR...TO, it assigns the starting value to the counter variable and stores the limiting and increment values. It also stores the line number of the FOR...TO command.

If you leave out STEP and its accompanying increment value, the program uses a default increment value of 1. If you specify STEP with an increment value larger than one (or a fractional increment value), the loop counter increases by that increment with each repetition. Negative increment values allow you to count backwards if your starting value is greater than your limit value. You can determine how many times a loop executes by using this formula:

$$\frac{\text{limit} - \text{start}}{\text{increment}} + 1$$

(See the NEXT command for a description of loop nesting and a programming example).

GOSUB

GOSUB <line number>

The GOSUB command jumps program execution to a *subroutine*. A subroutine is a section of the program that executes until a RETURN command is found. The RETURN command transfers control back to the statement that immediately follows the GOSUB command. If GOSUB is on a line by itself, execution returns to the next line. If it is part of a multiple statement line and another statement follows it, program execution returns to the next statement in the line.

A subroutine is a group of program lines that can be repeated many times in a program. To save space and entry time, make a repeated procedure into a subroutine and place it at the end of a program. Each time you want to use that procedure, use a GOSUB command followed by the first line number of the subroutine. You should separate subroutines from the main body of the program with an END command.

You can *nest* subroutines; that is, you can place one subroutine within another up to several levels. ABasiC keeps track of the levels of subroutine by the sequence of GOSUBs it finds. As the program encounters RETURN commands, it returns execution to the place it saved for the most recent GOSUB.

Make sure there's one RETURN command for every GOSUB you use in a program so ABasiC doesn't confuse the order of execution. Having too few RETURN commands fails to bring program execution back to the first GOSUB location. Having too many RETURN commands causes an error.

```
10 FEET = 35: INCHES = 3
30 GOSUB 1000
40 FEET = 2: INCHES = 11
60 GOSUB 1000
70 FEET = 6: INCHES = 5
90 GOSUB 1000
100 END
1000 TOTINCHES = FEET*12 + INCHES
1010 METERS = TOTINCHES/39.37
1020 PRINT "LENGTH IS ";METERS;" METERS"
1030 RETURN
```

When you run the program, the result is:

```
LENGTH IS 10.7442 METERS
LENGTH IS .88900 METERS
LENGTH IS 1.9558 METERS
```

GOTO

GOTO <line number>

You can use the GOTO command to skip ahead in a program by specifying a larger line number than the one containing the GOTO statement. If you instruct the program to GOTO a line number that isn't in the program, an error results.

You can also use GOTO to create an endless loop by specifying the GOTO command's line number or a smaller line number. This kind of loop continues to execute until you break out by pressing Ctrl–C.

```
10 PRINT "YOUR MOTHER WEARS ";
15 GOTO 30
20 PRINT "PINK ARMY BOOTS UNDER HER ";
30 PRINT "STYLISH DRESSES."
```

When you run the program, the result is:

```
YOUR MOTHER WEARS STYLISH DRESSES.
```

```
10 PRINT "FOREVER ";
20 PRINT "AND EVER "
30 GOTO 20
```

When you run the program, the result (partially shown) is:

```
FOREVER AND EVER
AND EVER
AND EVER
AND EVER
AND EVER
AND EVER
```

and so on, until you press the Ctrl–C.

IF...GOTO

```
IF <relationship> GOTO <line number>
```

IF...GOTO is a form of IF...THEN. For more information on IF...GOTO, read the description of IF...THEN.

```
IF X > 3 GOTO 300
```

IF...THEN...[ELSE]

```
IF <relationship> THEN [GOTO] <line number>
```

(or)
```
IF <relationship>  THEN  <statement> [:<statement>]...
```

(or)
```
IF <relationship>  THEN  <statement> [:<statement>]...
    ELSE <statement> [:<statement>]...
```

IF...THEN is a conditional branching command. It tests the truth of a relationship; depending on the outcome, it then *branches,* or forks, to different parts of the program. IF...THEN gives a computer the ability to make decisions and to react differently to different circumstances.

An IF...THEN statement usually contains a relationship between the IF and the THEN. A conditional action follows THEN; this action might be a command, a series of commands, or a line number. A line number must stand alone or with only GOTO, as in the first form of the syntax. Any statements that belong to the conditional action must fit on the program line with the IF...THEN statement. The condition does not affect the program lines that follow.

Optionally, you can use the ELSE keyword to list statements that execute if the specified relationship is *not* true. For example, the following statement:

```
IF A=0 THEN GOTO 40 ELSE B = B+15
```

tests the value of A. If A equals zero, program control goes to line 40. If A is not zero, B's value is increased by 15.

When ABasiC executes an IF...THEN command, it evaluates the relationship as true or false. If the relationship is true, the conditional actions are executed, and the program continues. If the relationship is false, program execution goes to the next line, and ABasiC ignores all the conditional actions remaining on the IF...THEN program line.

If the conditional action following "THEN" is a line number and the relationship is true, the program execution jumps to that line. If the relationship is false, the program continues with the line immediately following the IF...THEN.

To make the conditional jump more apparent, you can substitute "GOTO" for the element "THEN" to create an IF...GOTO statement. IF...GOTO works just like an IF...THEN statement, except after GOTO you can't use commands as a conditional activity. You can only list a line number.

When an IF...THEN statement evaluates a relationship, it substitutes −1 for the relationship if it's true and 0 if it's false. It then performs the conditional activity if the result is a nonzero value. You can place a numeric expression, such as

```
X MOD 3
```

instead of a relationship between "IF" and "THEN." IF...THEN tests the expression for true (nonzero) or false (0) and proceeds accordingly.

```
10 INPUT NAME$
20 IF NAME$="BALZNAR" THEN 100
30 PRINT "HI THERE, "; NAME$
40 END
100 PRINT "HI, BALZNAR.  YOU OWE ME TWENTY DOLLARS."
110 PRINT "I ACCEPT MAJOR CREDIT CARDS."
```

When you run the program, the result might be:

```
?RAPUNZEL                                          (Entered by user)
HI THERE, RAPUNZEL
```

When you run it again, the result might be:

```
?BALZNAR                                           (Entered by user)
HI, BALZNAR.  YOU OWE ME TWENTY DOLLARS.
I ACCEPT MAJOR CREDIT CARDS.
```

NEXT

NEXT [<counter variable> ,<counter variable>]...

The NEXT command marks the end of a program loop. The FOR...TO command merely establishes a counting variable and sets loop limits. The NEXT command actually does all the counting and looping. If the computer encounters a NEXT with no FOR...TO command, it generates an error.

The counter variable is optional; if you omit it, ABasiC connects the NEXT command with the last FOR...TO statement it encountered. However, your code is more readable when you include the variable name.

You can nest FOR...TO...NEXT loops, one inside the other, up to a maximum depth of nine loops. Each loop begins with a FOR...TO command and ends with a NEXT command. A different counter variable must be used for each loop.

You can end several nested loops with one NEXT command by listing the counters in order, from the innermost to the outermost loop. For example,

if a loop using the counter K is inside a loop using the counter J, you can end both loops with the following command:

```
NEXT K, J
```

```
10 FOR I = 1 TO 4
20 PRINT "TIME #";I
30 NEXT I
```

When you run the program, the result is:

```
TIME #1
TIME #2
TIME #3
TIME #4
```

The following example illustrates two nested loops, with NEXT commands combined:

```
10 PRINT "SEGMENTS OF FIVE"
20 FOR I = 1 TO 5
30 PRINT
40 FOR J = 4 TO 0 STEP -1
50 PRINT (5*J)+I;" ";
60 NEXT J,I
```

When you run the program, the result is:

```
SEGMENTS OF FIVE
21  16  11   6   1
22  17  12   7   2
23  18  13   8   3
24  19  14   9   4
25  20  15   16  5
```

ON ERROR GOSUB

```
ON ERROR GOSUB <line number>
```

Use this form of the ON ERROR... statement to send execution to a subroutine. Otherwise, this form of the command works like the ON ERROR GOTO command (see below).

ON ERROR GOTO

```
ON ERROR GOTO <line number>
```

Use the ON ERROR command to prevent ABasiC from stopping execution due to an error. You can use this command to screen out improper input while the program runs. Follow the ON ERROR with a GOTO and a line number. (Alternatively, you may use the GOSUB command.)

When ABasiC encounters an ON ERROR GOTO statement, it notes the GOTO and the line number. If an error occurs during the program execution, the program does not stop and print out an error message; instead, it transfers control to the line number you specified.

Once ON ERROR... has been triggered, another error stops the program and produces an error message. You can turn off ON ERROR... by entering:

```
ON ERROR GOTO 0
```

If your program contains more than one ON ERROR... statement, only the last one executed remains in effect.

```
10 ON ERROR GOTO 1000
20 INPUT "DIVIDE BY WHAT NUMBER" ,A
30 X=100/A
40 PRINT "YOUR ANSWER: " ;X
50 ON ERROR GOTO 50000
60 END
1000 IF ERR = 11 THEN PRINT "SORRY, ";A;
     " WON'T WORK.":PRINT "PLEASE TRY ANOTHER NUMBER."
1020 RESUME
```

When you run the program, you could get the following result:

```
DIVIDE BY WHAT NUMBER?0                    (entered by user)
SORRY, 0 WON'T WORK.
PLEASE TRY ANOTHER NUMBER.
DIVIDE BY WHAT NUMBER?4
YOUR ANSWER: 25
```

Note the use of the variable ERR in line 1000. ERR is the ABasiC reserved word that contains the number of the last error that occurred.

ERL is another reserved word variable; its value is the line number of the most recent error. (See the programming example of the RESUME command to see how ERL is used.)

ON...GOSUB

```
ON <numeric variable> GOSUB <line number>
   [,<line number>]...
```

ON...GOSUB is similar to the ON...GOTO command, except that the line numbers branch to subroutines. Then ABasiC returns to normal program execution. Each line number must be the first line of a subroutine. When program execution returns, it picks up at the first command following ON.

```
10 INPUT "HOW OLD ARE YOU";AGE
20 PRINT "PEOPLE ";
30 ON INT(AGE/10)+1 GOSUB 100,200,300,400,500,600,
   700,800,900,1000
40 PRINT "SHOULD KNOW BETTER THAN TO ANSWER QUE
   STIONS ASKED BY A COMPUTER."
50 END
100 PRINT "UNDER THE AGE OF 10 ";:RETURN
200 PRINT "IN THEIR TEENS ";:RETURN
300 PRINT "IN THEIR TWENTIES ";:RETURN
400 PRINT "IN THEIR THIRTIES ";:RETURN
500 PRINT "IN THEIR FORTIES ";:RETURN
600 PRINT "IN THEIR FIFTIES ";:RETURN
700 PRINT "IN THEIR SIXTIES ";:RETURN
800 PRINT "IN THEIR SEVENTIES ";:RETURN
900 PRINT "IN THEIR EIGHTIES ";:RETURN
1000 PRINT "IN THEIR NINETIES ";RETURN
```

When you run the program, the result might be:

```
HOW OLD ARE YOU?53          (Input by user)
PEOPLE IN THEIR FIFTIES SHOULD KNOW BETTER THAN TO ANSWER
QUESTIONS ASKED BY A COMPUTER.
```

ON...GOTO

```
ON <integer variable> GOTO <line number>
[,<line number>]...
```

ON...GOTO is a multiple branching command; it can branch to many different lines using a single statement. The number of branches it uses is limited by the number of line numbers that fit on a single program line.

The value of the numeric variable that follows ON determines which line number control branches to. The line numbers after GOTO are the different branches. You should assign the variable's value before the program reaches the ON...GOTO command.

When an ON...GOTO executes, ABasiC rounds the value of the variable to an integer value, if necessary. ON...GOTO uses that value to branch to the desired line number. If the value is 1, ON...GOTO sends control to the

first line number listed. If the value is 2, it sends control to the second line number, and so on.

ABasiC ignores the ON...GOTO command if the variable value is less than 1 or is greater than the number of branches. It continues with the next statement.

You can replace GOTO in the ON...GOTO command with GOSUB (see below). ON...GOTO and ON...GOSUB can do the work of several IF...THEN commands. The trick to using them well is to reduce the variable value to an integer in the appropriate range. This tightens up your program, saves memory, and speeds program execution.

```
10 INPUT "WHAT IS YOUR CHOICE (1-5)";CHOICE
20 ON CHOICE GOTO 100,200,300
100 PRINT "YOU CHOSE VANILLA ICE CREAM WITH CHOCOLATE
    TOPPING.":END
200 PRINT "YOU CHOSE TUTTI-FRUTTI ICE CREAM WITH
    MAPLE SYRUP.":END
300 PRINT "YOU CHOSE TUNA FISH ICE CREAM WITH SARDINE
    TOPPING.":END
```

When you run the program, the result might be:

```
WHAT IS YOUR CHOICE (1-5)?2          (Input by user.)
YOU CHOSE TUTTI-FRUTTI ICE CREAM WITH MAPLE SYRUP.
```

RESUME

RESUME [NEXT | 0 | <line number>]

Use the RESUME command to continue program execution after an error has been trapped (that is, caught by your program instead of stopping execution) using the ON ERROR GOTO/GOSUB command. You can only use RESUME in an error trapping routine; otherwise, an error is generated that stops your program.

When you use RESUME alone or with an argument of 0, it sends program control back to the first statement on the line in which the error occurred.

The programming example for the ON ERROR GOTO command illustrates this.

When RESUME is followed by the NEXT keyword, control transfers to the line immediately following the one that caused the error. If a line number follows RESUME, program execution transfers to the specified line.

```
10 ON ERROR GOTO 1000
...
1000 IF ERL < 150 THEN RESUME 400
1010 IF ERL < 250 THEN X = 0: RESUME 600
1020 RESUME NEXT
```

RETURN

RETURN

Use the RETURN command to mark the end of a subroutine in a program. RETURN is always paired with a corresponding GOSUB command, which points to the beginning of the subroutine. When the computer encounters a RETURN, it jumps program execution back to the statement immediately following the most recent GOSUB. If ABasiC finds a RETURN command with no preceding GOSUB command, it produces an error. See the GOSUB command description for a program example using RETURN.

STOP

STOP

Use the STOP command to halt execution of a program and return the computer to immediate mode. When ABasiC encounters a STOP command in a program, it responds:

```
Stop at line n
Br
```

where n is the program line number that contains the STOP command. ABasiC preserves any variable values when the STOP occurs. It also keeps

any open files as they are. They aren't cleared or closed when STOP executes.

You can resume program execution after the STOP command by typing

```
CONT
```

The computer remembers where it stopped execution and will continue at the statement immediately following the STOP command. If you want to start the program at a line other than the stopping point, use

```
GOTO n
```

where n is the number of the line at which you wish to resume execution. Alternatively, you can use RUN to restart the program from the beginning.

Since STOP breaks the computer out of a running program, you won't often use it in a finished program. However, it is helpful in debugging a program. If you insert STOP in a section of a program where you suspect a problem, you can halt execution at that point. Then use immediate mode commands to check the status of variables and files. Once you have the information you need, you can restart the program with CONT.

```
10 PRINT "SOMETIMES THE SPRINTER ";
20 PRINT "DOESN'T MAKE IT TO "
25 STOP
30 PRINT "THE END.
```

When you run the program, the result is:

```
SOMETIMES THE SPRINTER DOESN'T MAKE IT TO
Stop at line 25
Br
```

Once you enter CONT, the program continues:

```
THE END
```

WEND

WEND

Use the WEND command to end an indefinite loop. The WHILE statement sets up a condition that is a relationship. The program tests the truth of the relationship and executes all statements between the WHILE statement and the next WEND it finds. Once the relationship's value is false (0), program control jumps to the statement immediately following the WEND command.

(See the description of WHILE for a programming example that uses WEND.)

WHILE

WHILE <relationship>

The WHILE command works together with WEND (While END) to control an *indefinite loop*. An indefinite loop simply means that the number of times a group of statements repeat execution is conditional, rather than fixed, as with the FOR...TO...NEXT structure. The relationship sets up a condition that makes a group of statements execute repeatedly until the WHILE statement detects that the condition is no longer true.

An indefinite loop is helpful when you want the program to perform certain actions on a variable-length set of data, such as numbers in a disk file. The general form is:

```
WHILE <relationship> <statement>
[:<statement>...]
WEND
```

When program execution reaches the WHILE statement, it tests the truth of the relationship. If the relationship returns a true (-1), all statements that follow the condition execute. As soon as ABasiC comes to the WEND statement, control jumps back to the WHILE statement, which tests the truth of the relationship again. Once the value is false (0), program control jumps to the statement immediately following the WEND statement.

You can also *nest* WHILE...WEND loops (that is, place one loop inside another which is inside yet another, etc.). As soon as a program that contains nested WHILE...WEND loops reaches a WEND command, it transfers control back to the last WHILE statement it encountered.

```
10 WHILE NAME$ <> "JOHN"
20 INPUT NAME$
30 PRINT "HELLO, ";NAME$
40 WEND
50 PRINT NAME$; ", AT LAST!"
```

When you run the program, the result might be:

```
?MARINA
HELLO, MARINA
?ALFRED
HELLO, ALFRED
?JOHN
JOHN, AT LAST!
```

Functions

This chapter describes arithmetic, trigonometric, and string functions in ABasiC. See "Input/Output Commands" and "Graphics Commands" for functions specific to these categories.

Arithmetic Functions

In addition to the simple arithmetic operations, such as addition and exponentiation, ABasiC provides several built–in, special numeric functions for your programming convenience.

ABS

`ABS(<numeric expression>)`

The ABS (ABSolute value) function returns the absolute value of any numeric expression. Positive expressions and those equal to zero are left unchanged. Negative expressions are multiplied by −1 to make them positive. (Using the ABS function with the maximum negative integer causes overflow.)

```
PRINT ABS(.345)

returns

.345
```

```
PRINT ABS(-35)
```

returns

```
35
```

CDBL

CDBL (<numeric expression>)

The CDBL function converts the value of any numeric expression to a double-precision number. For example, you can use CDBL to increment the value of a double-precision variable by an integer amount or zero.

```
30 A  = 123.0012: B = 89.98933
40 R#= CDBL(A)*CDBL(B)
50 S#= CDBL(A*B)
60 PRINT R#: PRINT S#
```

Line 40 performs multiplication in double precision, while line 50 multiplies in single precision and then converts to double precision. The result is as follows:

```
11058.95462619
11058.955078125
```

CSNG

CSNG (<numeric expression>)

The CSNG function converts the specified numeric expression to a single-precision number.

CINT

CINT (<numeric expression>)

The CINT function converts the specified numeric expression to an integer, rounding it if necessary.

DEC

DEC ("string")

The DEC function converts any numeric string to a decimal number. The string can include the hex letters A to F. Hexadecimal values must be in the range of 0 to FFFF.

```
PRINT DEC("&H2A")

returns

42
```

EXP

EXP(<numeric expression)

The EXP (EXPonential) function raises the value of *e* (2.71828183) to the power specified by the numeric expression. EXP is the opposite of the LOG function. Using a very large argument with EXP produces an error if the result is larger than the allowed limit for floating point numbers. Fractional and negative numbers cause no overflow.

```
PRINT EXP(40)

returns

2.35385e+17
```

```
PRINT EXP(30-41.5)

returns

1.01301e-05
```

FRE

FRE [(<numeric expression>)]

The FRE function returns the number of free bytes currently in the largest contiguous section of RAM. These bytes are available for additional program lines, variables, and arrays.

It's a good habit to use FRE often while you enter program lines to keep track of the amount of remaining RAM.

FIX

FIX (<numeric expression>)

The FIX function converts a numeric expression to an integer by truncating any fractional portion of the expression's value. FIX differs from the INT function (described below) in that INT rounds the expression's value to the next lower integer.

```
FIX (37.605)
```

returns

```
37
```

INT

INT (<numeric expression>)

Use the INT (INTeger) function to convert a numeric expression to an integer by rounding it down to the next lower integer, whether positive or negative.

```
PRINT INT(15.7864)

returns

16
```

```
PRINT INT(-456.591)

returns

-457
```

Note: If you want a function that works like INT but rounds negative numbers to the next *higher* integer, try the following:

```
SGN(X)*INT(ABS(X))
```

where X is the number you want to round. Better yet, use the DEF FN command to create your own function using this expression.

LOG

LOG(<positive numeric expression>)

The LOG (LOGarithm) function calculates the natural logarithm of a numeric expression. LOG is the opposite of the EXP function, because LOG returns the power of *e* (2.71828183) that creates the specified numeric expression. Negative expressions produce an error, because there are no powers of *e* that yield a negative number.

```
PRINT LOG(835)

returns

6.72743
```

LOG10

LOG10(<positive numeric expression>)

The LOG10 function returns the base–10 logarithm of a number. The value of the numeric expression must be greater than zero.

```
10 X = LOG10(1000)
20 PRINT X

When you run the program ( since 10 ^3 = 1000), the result is:

3
```

MOD

<numeric expression> MOD <numeric expression>

The MOD (MODulo) function returns the remainder of integer division, where the left expression is divided by the right expression. For example,

```
28 MOD 3
```

returns

```
1
```

(the remainder after division).

RND

RND [(<numeric expression>)]

Use RND (RaNDom) to obtain a random fraction between 0 and 1. The numeric expression in parentheses, the argument, affects the way RND operates in the following way:

If the argument is a *positive* number, RND returns the next number in the current sequence of random numbers. The effect is the same if you leave out the argument altogether.

If the argument is a *negative* number, RND reseeds the random number generator (that is, uses the specified argument to generate a new sequence of numbers) and returns the first number in the new sequence.

If the argument is equal to *zero*, RND returns the last number generated without affecting the current sequence. RND(0) is useful for debugging a program when you want a consistent value returned.

```
10 FOR I=1 TO 4
20 X= RND(8):PRINT X
30 Y=INT(100*X):PRINT Y
40 NEXT I
```

When you run the program, the result might be:

```
.185564016
18
.0468986348
4
.827743801
82
.554749226
55
```

See also the RANDOMIZE command.

SGN

SGN(<numeric expression>)

The SGN (SiGN) function determines whether a numeric expression is positive, negative, or equal to zero. If positive, SGN returns a 1; if negative, it returns a −1; if equal to 0, it returns a 0.

```
A = -4.353e21: PRINT SGN(A)
```

returns

```
-1
```

```
A = 375: PRINT SGN(A)

returns

1
```

SQR

SQR(<positive numeric expression>)

The SQR (SQuare Root) function takes the square root of the numeric expression following it. The square root of a negative number is not a real number; thus, if you attempt to use SQR with a negative number you produce an error. (The SQR function works with 0.)

```
PRINT SQR(34.5)

returns

5.87367
```

VARPTR

VARPTR(<variable> | #<file number>)

The first form of the VARPTR function (with a variable) returns the address of the specified variable. You can use such an address as a value to pass to a machine language routine (see "File Management Commands" for details). The address that VARPTR returns is an integer value that your program can POKE into another location.

You can also get the starting address of an array using the following syntax:

```
VARPTR( <variable>(0) )
```

Be sure to initialize all simple variables before you get the address of an array, because the address of an array can change whenever the system assigns a memory location to a variable.

The second form of VARPTR (with # and a file number for an argument) returns the address of the input/output file buffer for the file you assigned to this number. (See the OPEN command for details on assigning file numbers.) You can only use this form of the command with random access data files.

```
20 D$ = "03/26/75"
...
50 ADDR = VARPTR(D$)
60 PRINT ADDR
```

When you run the program, the result might be:

```
264942
```

Trigonometric Functions

ABasiC offers four trigonometric functions to calculate sines, cosines, tangents, and arctangents for different angles. These trig functions work with values expressed in radians: *pi*/2 instead of 90 degrees; *pi*/4 instead of 45 degrees; *pi* instead of 180 degrees; and so on. If you're more comfortable working with degrees than with radians, you can convert a function's output to degrees by multiplying by 180/*pi*. Likewise, you can convert degrees back to radians by multiplying by *pi*/180.

PI is a system variable that you can use in programs. It is a single–precision variable with a value of 3.14159.

If you need more trigonometric functions than the four included here, you can create them using the table (titled Derived Trigonometric Functions) that follows the trigonometric command function descriptions. For greater efficiency, use the expressions in the table to define new functions with the DEF FN command.

The general function form is:

```
<function name> (<numeric expression in radians>)
```

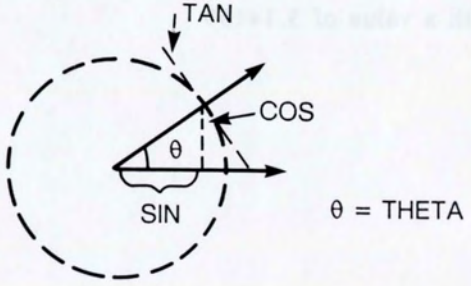
ATN Returns the ArcTaNgent of the numeric expression in parentheses. The arctangent is the opposite of the TAN function, since it calculates the angle (in radians) whose tangent equals the numeric expression. ATN always returns values in the range of $-pi/2$ to $+pi/2$.

COS Returns the cosine of the numeric expression in parentheses.

SIN Returns the sine of the numeric expression in parentheses.

TAN Returns the tangent of the numeric expression in parentheses. If the expression is equal to $pi/2$ (90 degrees), $3*pi/2$ (270 degrees), or another value equivalent to these two angles, an error occurs. This is because the tangent of an angle is the cosine divided by the sine, and the sine of the specified angles is zero.

The following programming example demonstrates each of the four trig functions. Suppose you want your program to calculate the function values for a 30-degree angle, THETA:



```
10 THETA = 30*PI/180: REM radian value = .5236
20 X = COS(THETA): Y = SIN(THETA)
30 TANGENT = TAN(THETA)
40 PRINT X,Y,TANGENT
50 PRINT ATN(.5774)
```

When you run the program, the result is:

```
.5000      8660      5774
30
```

If you need advanced trigonometric functions that ABasiC does not offer, you can create them using the following expressions:

Derived Trigonometric Functions

Secant:	1/COS(X)
Cosecant:	1/SIN(X)
Cotangent:	1/TAN(X)
Inverse sine:	ATN(X/SQR(-X*X+1))
Inverse cosine:	-ATN(X/SQR(-X*X+1))+PI/2
Inverse secant:	ATN(X/SQR(X*X-1))
Inverse cosecant:	ATN(X/SQR(X*X-1))+ (SGN(X)-)*PI/2
Inverse cotangent:	ATN(X)+PI/2
Hyperbolic sine:	(EXP(X)-EXP(-X))/2
Hyperbolic cosine:	(EXP(X)+EXP(-X))/2
Hyperbolic tangent:	EXP(-X)/(EXP(X)+EXP(-X))*2+1
Hyperbolic secant:	2/(EXP(X)+EXP(-X))
Hyperbolic cosecant:	2/(EXP(X)-EXP(-X))
Hyperbolic cotangent:	EXP(-X)/(EXP(X)-EXP(-X))*2+1

Inverse hyperbolic sine:	LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine:	LOG(X+SQR(X*X-1))
Inverse hyperbolic tangent:	LOG((1+X)/(1-X))/2
Inverse hyperbolic secant:	LOG((SQR(-X*X+1)+1)/X)
Inverse hyperbolic cosecant:	LOG((SGN(X)*SQR(X*X+1)/X)
Inverse hyperbolic cotangent:	LOG((X+1)/(X-1))/2

String Functions

It's often useful to change portions of a string variable's current value, or to compare parts of one string with another. ABasiC provides a number of special operations you can easily perform on strings and string variable values in your programs. (See also Screen Position Functions and the REPLACE$ command.)

ASC

`ASC("string")`

The ASC function turns a string character into an ASCII code number; it is the inverse of CHR$. You can enter a string of any length, although ASC returns only the ASCII code of its first character. For example,

```
PRINT ASC("BANANAS")
```

returns the ASCII code of the first character, B. Appendix B contains an ASCII code list of the ABasiC character set.

CHR$

CHR$(<integer>)

The CHR$ function converts an integer into a one-character string according to the ASCII code. The range for the number is 0 to 255. For example, the ASCII code for "A" is 65, so the statement:

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

returns

A

Appendix B lists the ASCII code table.

CHR$ is convenient for inserting hard to handle characters into a string. For example, you can use CHR$ to add a line feed and carriage return--represented by ASCII 10--to a string of characters. You can't put a carriage return in a string without CHR$, because when you press Return it immediately terminates the string you are entering.

```
10 A$="STAIR"+CHR$(10);SPACE$(5);"STEP"
20 PRINT A$
```

When you run the program, the result is:

```
STAIR
     STEP
```

HEX$

HEX$(<numeric expression>)

The HEX$ function returns a string that is the hexadecimal (base-16) representation of the numeric expression you specify. The letters A through F represent the decimal digits from 10 to 15. HEX$ does not add a

leading "&H" to the string to represent the base. It rounds the value of the numeric expression to the nearest integer before converting it.

INSTR

INSTR([<integer> ,] "target string", "pattern string")

The INSTR function searches for one string (the pattern string) within another (the target string) and returns the position of the pattern string. The target and pattern strings can be string constants, expressions, or variables. The integer is an optional starting point within the target string, and must have a value between 1 and 255.

```
10 X$ = "TO BE OR NOT TO BE"
20 X = INSTR(6,X$,"BE")
30 PRINT X
```

When you run the program, the result is:

```
17
```

The first occurrence of "BE" is not returned, since the starting point is the sixth character.

LEFT$

LEFT$("string",<integer>)

Use the LEFT$ function to extract a substring (a portion of a string), starting from the left side of a larger string. After the LEFT$ command, place the target string or variable name in parentheses, followed by a comma and the number of characters you want to extract. This number must be between 1 and 255. A higher or lower integer produces an error. (Fractional numbers are simply truncated to the lower integer.)

If you specify a substring length equal to or greater than the original string, the function returns the entire original string. If you specify a length of zero (0), the result is a null string (containing no characters).

```
10 Z$="YOU"
20 FOR I=1 TO 3
30 PRINT LEFT$(Z$,I)
40 NEXT I
```

When you run the program, the result is:

```
Y
YO
YOU
```

LEN

LEN("string")

The LEN function returns the number of characters (including blanks, punctuation marks, and nonprinted characters) in a string. ABasiC does not count the enclosing quotation marks. After the LEN function name, enclose the string or string variable in parentheses.

```
10 A$="POLYSYLLABIC"
20 PRINT LEN(A$)
```

When you run the program, the result is:

```
12
```

MID$

MID$("string",<integer-1> [,<integer-2>])

MID$ also extracts substrings. Unlike LEFT$, however, it can begin at any location within the string. You must specify the beginning character number of the target string (integer–1). You can also give the length of the desired substring (integer–2). If you omit the length, MID$ returns the entire remainder of the original string, beginning with the requested character.

Enclose the string or string variable you want in parentheses, followed by a comma and the beginning character number (from the left) of the substring. If you want to specify a substring length, add another comma followed by the number of characters.

If the starting character number is greater than the length of the main string, or if you specify a substring length of zero, the MID$ function returns a null string (containing no characters). If you specify a substring length that's longer than the remaining number of characters in the target string, the substring includes the remainder of the target string.

```
10 A$="MISTER BOZNO"
20 GOSUB 100
30 A$="MISTER STRUBNAR"
40 GOSUB 100
50 END
100 PRINT MID$(A$,8)
110 RETURN
```

When you run the program, the result is:

```
BOZNO
STRUBNAR
```

OCT$

OCT$ (<numeric expression>)

The OCT$ function returns a string that is the octal (base–8) representation of the numeric expression. ABasiC rounds the value of the numeric expression to the nearest integer before it converts it.

RIGHT$

RIGHT$ ("string",<integer>)

The RIGHT$ function works like LEFT$, but it extracts a substring starting from the *right* side instead of the left side of the target string. Place the target string (or variable name) within parentheses, followed by a comma and the number of characters you want to extract.

```
10 JOG$="THE MAN WITH THE BIG NOSE IS RUNNING."
20 PRINT RIGHT$(JOG$,25)
```

When you run the program, the result is:

```
THE BIG NOSE IS RUNNING.
```

SPACE$

SPACE$ (<integer>)

The SPACE$ function embeds blanks within a PRINT statement. The integer you specify determines how many spaces to print. It must be between 0 and 255. (Note: The SPC function is more efficient for this purpose and its use is recommended. See the subsection on Screen Position Functions for details.)

You can also use SPACE$ outside of a PRINT statement. For example, the following is a valid assignment statement:

```
A$ = "Subtotal"+SPACE$(4)+"$"
```

STRING$

STRING$(<integer-1> ,<integer-2> | "string")

The STRING$ function returns a string of given length filled with the specified characters. The first integer specifies the length, and must be between 0 and 255. The second argument, either a second integer or a string expression, defines the character(s) with which to fill the string.

If the second argument is numeric, ABasiC interprets its value as the ASCII code for the fill character. If the argument is a string, it must be at least one character long. However, if the string is more than one character, ABasiC uses only the first character to fill the string.

```
10 STAR$ = STRING$(20, "*")
20 PRINT STAR$:PRINT "STAR-STUDDED PROGRAM":PRINT STAR$
```

When you run the program, the result is:

```
********************
STAR-STUDDED PROGRAM
********************
```

STR$

STR$(<numeric expression>)

Use the STR$ function to convert a numeric expression to a string expression. This is helpful when you want to manipulate the digits in a number as string characters. The numeric expression can be any valid variable or constant in ABasiC, including integers or floating point numbers. You can express a floating point number as a simple number or in scientific notation.

When STR$ returns the string conversion of a positive number or zero, it adds a blank at the beginning of the string. STR$ returns the string conversion of negative numbers with a minus sign in place of the preceding space. Simple numbers of more than nine digits convert to scientific notation before becoming a string. If scientific notation numbers can be reduced accurately to a simple number of nine digits or less, they convert to a simple number before becoming a string.

The STR$ function operates with decimal values. Use HEX$ to convert hexadecimal values and OCT$ for octal values.

```
10 A=105.78
20 C$=STR$(A)
30 CENT$=RIGHT$(C$,2)
40 PRINT CENT$;" CENTS"

When you run the program, the result is:

78 CENTS
```

VAL

VAL("string")

Use the VAL function to convert a string to a number. Following VAL, put the string you want to convert in parentheses. The VAL function is the opposite of STR$ in that it turns a string into a number rather than a number into a string.

The first character in the string must be a digit, a plus sign, or a minus sign. It can also be a blank, because VAL ignores blanks in strings. If it finds any other first character, VAL returns a value of zero.

If VAL finds an acceptable first character, it converts until it either comes to the end of the string or encounters a non–numeric character. For example:

```
PRINT VAL("435 ELM STREET")
```

returns

```
435
```

VAL also converts a string in correct scientific notation form to a number. For example:

```
PRINT VAL("1.756E18")
```

returns

```
1.756E+18
```

```
10 ADDRESS$="ST. DISMAS, CA  94516"
20 ZIP$=RIGHT$(ADDRESS$,5)
30 IF VAL(ZIP$)>94000 THEN PRINT "NORTHERN CALIFORNIA"
```

When you run the program, the result is:

```
NORTHERN CALIFORNIA
```

Graphics Commands

The graphics commands let you draw shapes and fill areas of the display with colors and patterns.

You normally draw lines, circles, boxes, etc., on the screen with a primary pen, called PENA. A secondary pen, PENB, fills in background colors and performs other special tasks. Certain closed figures on the screen can have an outline in a third color, which is drawn by PENO (O for outline).

Several graphics commands refer to the *pixel cursor* coordinates; don't confuse these with the *character* coordinates that several nongraphics commands and functions use. (There are normally 320 pixels and 40 characters per display line.) In other words, each character is eight pixels wide. The screen can display up to 23 lines of text and contains 200 scan lines. You can think of a scan line as the vertical counterpart to a pixel. Each text line is eight scan lines high.

The number of colors that are available depends on the display *depth*. The default depth is four, allowing 2^4, or 16, colors. There are 32 color registers. The table entitled "Default Color Register Values" at the end of "Graphics Commands" describes the red, green, blue composition of the default colors assigned to the first 16 of these registers. You can change the display depth to allow 32 different colors, but the upper 16 colors are just the same as the lower 16. You can redefine their color composition with the RGB command.

Graphics pixel coordinates have their origin, (0,0), in the upper left corner of the output window. The x pixel cursor coordinate increases from left to right, and the y coordinate increases downward. Information regarding the current cursor location always refers to the program output window (usually the entire monitor screen). When you use several windows at once, the information refers to the currently active window.

ANIMATE

ANIMATE <array%>, <x%>, <y%>, <view%>

The ANIMATE command controls the position and view sequence of each of the movable graphics objects that you've loaded into the specified integer array. These graphics objects are loosely referred to as *sprites*. Associate each ANIMATE command you use with a particular sprite by specifying the sprite's array name.

Follow the array name with an argument list separated by commas. This list consists of the coordinates of the sprite's upper left corner, x% and y% (given in pixels) and the view number that you want displayed. The file that you saved after editing your sprite keeps track of the sprite's size and number of views. If you specify view number 2, ABasiC retrieves the number of bytes per view and goes to the appropriate array element to fetch the information to display.

Animation on an Amiga is similar to movie animation: By rapidly replacing one view of your sprite with another at the same screen location, you create the illusion of motion.

```
20 DIM KAT1%(340)
30 BLOAD KAT,VARPTR(KAT1% (0))
...
100 ANIMATE KAT1%(), 200, 50, 0
```

When you run the program, ABasiC loads the KAT sprite file into array KAT1%() and displays view 0.

AREA

AREA([TO] x1,y1 [TO x2,y2...])

The AREA command defines and flood fills a closed area on the screen. Each area must include at least three points. If the keyword TO is the first item in parentheses, ABasiC uses the current pixel cursor position as the first point. If the figure you want to create has more vertices than can fit on a logical line, use the MAT AREA command to define the points as array elements. (MAT AREA is also a good choice if you want to define an area once and use it several times.)

When AREA executes, it automatically connects the last point specified to the first point and fills in the region. (See the DRAW command for making *open* line drawings.) AREA uses the fill pattern most recently defined by the PATTERN command. PATTERN uses the colors currently assigned to PENA and PENB.

You can also choose whether or not to have a visible outline by setting the value of the system variable OUTLINE. To get a visible outline for a particular shape, include the following statement before you execute the AREA statement.

```
OUTLINE 1
```

If OUTLINE is 1, ABasiC uses PENO's current color and the current LINEPAT pattern to draw the outline. (The default LINEPAT pattern is a solid line, 16 "1" bits).

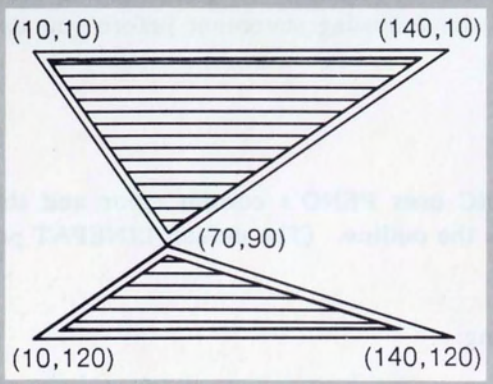
If you specify the following:

```
OUTLINE 0
```

the outline is invisible. The pattern simply fills to the border defined by the specified vertices.

```
10 PENA 7: PENB 6; PENO 12
15 FOR J = 0 to 3: READ SHAPE%(J): NEXT J
20 DRAWMODE 1: OUTLINE 1
30 PATTERN 4, SHAPE%()
40 AREA(10,10 TO 140,10 TO 70,90 TO 140,120 TO 10,120 TO 70,90)
100 DATA 0,0, 65535, 65535
```

When you run the program, the result is:



ASK CURSOR

ASK CURSOR <x%>, <y%>

The ASK CURSOR command provides you with the current cursor location. The horizontal position (in pixels) and the vertical position (in scan lines) is placed in the respective x% and y% integer variables that you specify.

```
10 ASK CURSOR HOR%, VERT%
20 PRINT HOR%, VERT%
```

ASK MOUSE

ASK MOUSE <x%>, <y%>, <b%>

The ASK MOUSE command provides information about the position of the mouse and whether the left button has been pressed since the last time it

was moved. (ABasiC has no way of detecting a click of the right button.) Your mouse should be in the left mouse port. The variables you specify receive the following values:

x% The horizontal position (in pixels) relative to the last currently active window in which the button was clicked.

y% The vertical position (in pixels) relative to the last currently active window in which the button was clicked.

b% The left button status since the mouse was moved last. Contains a 4 if the button was clicked and a 0 if not.

```
40 WHILE X1% < 340
50 GOSUB 200
60 ASK MOUSE X%, Y%, B%
70 IF B% <> 0 THEN GOSUB 450
80 WEND
```

ASK RGB

ASK RGB <integer>,<variable-1>,
<variable-2>,<variable-3>

The ASK RGB command gives you the color information in the color register indicated by the specified integer. This information includes the current combination of red, green, and blue colors assigned to that register. The corresponding values go in the variables you provide, which are represented as <variable-1> through <variable-3>.

For example, the default value for register 0 is:

red = 10, green = 8, and blue = 7

Thus, the following statement:

```
ASK RGB 0, R%, G%, B%
```

places the values of 6, 9, and 15 in variables R%, G%, and B%, respectively.

The integer you specify must be between 0 and 31, although only the first 16 registers have colors defined. Color register values out of range produce an error. See the table entitled "Default Color Register Values" at the end of this section for the default values of the first 16 color registers.

ASK WINDOW

`ASK WINDOW <width%>, <height%>`

Use the ASK WINDOW command to find out the size of the current output window. ASK WINDOW places the width (in pixels) in the first integer variable you specify and the height in the second integer variable. For details on ASK WINDOW, see "File Management Commands."

BOX

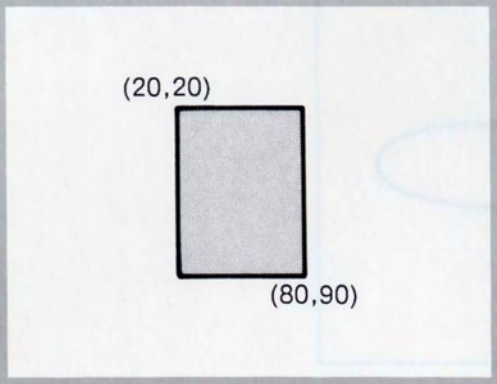
`BOX(x1,y1[;x2,y2]) [,<fill>]`

The BOX command draws a rectangle that has its upper left corner at x1, y1. You can optionally specify the coordinates x2, y2 to define the lower right corner. Otherwise, BOX uses the lower right corner of the current output window.

BOX uses PENO's current color and the current LINEPAT pattern to draw the box outline. The <fill> parameter should be 0 or 1 to indicate whether to fill the box with the current PATTERN (using the current PENA and PENB colors). If you specify 0, the box is hollow.

```
10 BOX( 20,20 ; 80,90) ,1
```

When you run the program, the result is:



CIRCLE

CIRCLE [(x,y)] ,<radius> [,<aspect>]

Use the CIRCLE command to draw a circle or an ellipse centered at pixel coordinates x,y. CIRCLE uses the current PENO color. However, it draws the circle in a solid line and ignores the current LINEPAT pattern.

If you don't specify coordinates, CIRCLE uses the current pixel cursor position. You must also specify the radius. The Amiga's horizontal (x) units are not proportional to its vertical (y) units. The radius you specify is interpreted as horizontal units.

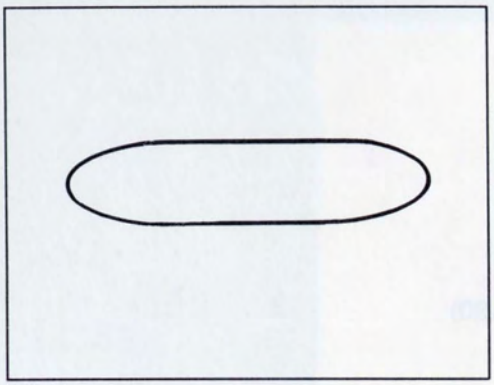
The optional aspect parameter is the ratio of height to width:

h/w

The default ratio is 1/1, or 1; this is a circle. An ellipse with a height half its width has an aspect of 1/2, or 0.5.

The following CIRCLE statement draws the figure below:

```
CIRCLE (30,120), 14, .25
```



If you want to fill in a circle with a pattern, execute the CIRCLE statement first. Then execute a PAINT statement with coordinates inside the circle.

DRAW

```
DRAW ( [TO] x1,y1 [ TO x2,y2...] ) [,<color>]
```

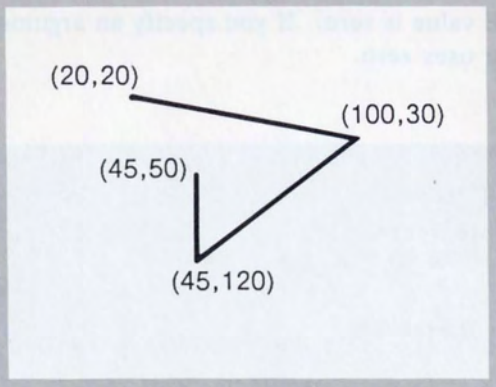
Use the DRAW command to draw a line between the specified points. DRAW displays a single point if you specify only one coordinate pair. Two coordinate pairs describe a line. To draw a line from the current pixel cursor position to the point (x1,y1), place the keyword TO in front of the first pair of coordinates.

If you place a comma and a color register number after the list of coordinate pairs, DRAW uses that color. Otherwise, it uses the color currently assigned to PENA. (If you want to use the same color with several DRAW commands, it's more efficient to assign the desired color to PENA first.) DRAW uses the current line pattern (see the LINEPAT command).

Three or more coordinate pairs describe an open shape. Unlike the AREA command, DRAW doesn't automatically connect the last point with the first. The number of vertices can't exceed the number of coordinate pairs you can fit on a single logical line. To make a more complex drawing, use the MAT DRAW command. (MAT DRAW is also a good choice if you want to define a shape once and use it several times.)

```
10 DRAW (20,20 TO 100,30 TO 45,120)
20 DRAW (TO 45,50)
```

When you run the program, the result is:



DRAWMODE

DRAWMODE <integer>

The DRAWMODE command sets the mode for line drawing, area fill, and text operations. Specify an integer between 0 and 2 as follows:

0 Replaces the color displayed at the drawing point with the current PENA color; changes only the affected pixels without disturbing the background.

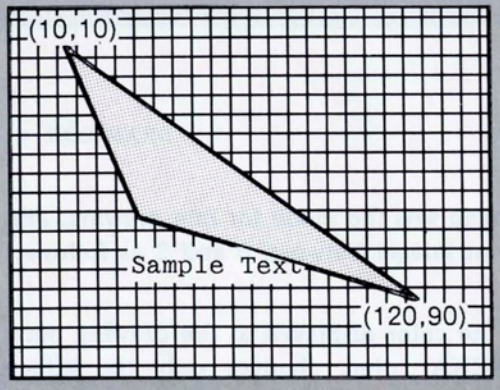
1 Completely overwrites the current screen display (or the current output window) with the PENA and PENB colors. For any two-color writing on the screen (for example, text with filled-in background or patterned area fill), replaces every "1" bit with PENA's color and every "0" bit with PENB's.

2 Draws on top of the background without blocking out what background shows through (that is, this mode performs an XOR--*exclusive or*--on the drawing point with the PENA color). If PENA's color is the same as the background, erases the drawing point.

The default DRAWMODE value is zero. If you specify an argument that is out of range, the program uses zero.

```
10 PENA = 2: PENB = 4
20 FONT = 1: DRAWMODE 0
30 PRINT AT(3,8); "Sample text"
40 AREA (10,10 TO 120,90 TO 30,60)
```

When you run the program, the result is:



See also the AREA command for a programming example of DRAWMODE 1.

FONT

FONT <integer>

The FONT command sets the active text font to the integer value you specify. The integer must be between 0 and 2:

0 Sets font at the current default size contained in the Preferences file. This is initially 40–column size (low–resolution). If you change the font size to 32–column in the Preferences file, the machine shows 32–column font size when you turn it on.

1 Sets font at 40–column size (40 characters per line). Character size is 8 by 8 pixels. (In high resolution, sets font at 80–column size.)

2 Set font at 32–column size. Character size is 10 pixels wide by 9 pixels high. (In high resolution, sets font at 64–column size.)

The default FONT value is zero and is initialized at 40-column size. If you specify an argument that is out of range, the program uses zero. See the DRAWMODE command for a programming example of FONT.

GSHAPE

GSHAPE (x,y) ,<array%> [,<mode>]

Use the GSHAPE (Get SHAPE) command to display a copy of a rectangular portion of the display you saved using the SSHAPE command. The coordinates (x,y) specify the pixel position of the upper left corner of the rectangle on the screen (or current output window).

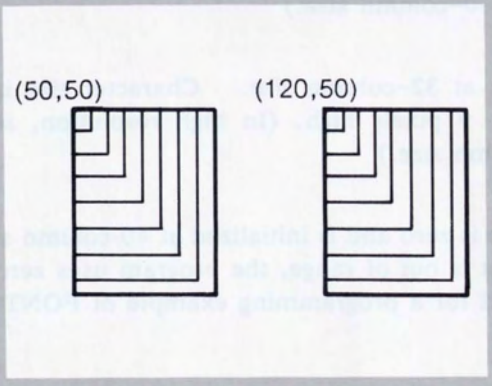
The SSHAPE command that saved the rectangle defines its size. SSHAPE saves the rectangle in the specified integer array; you then specify the same array name in the corresponding GSHAPE command.

The optional mode should be an integer from 0 to 2, one of the DRAWMODE values. (See the DRAWMODE command for a description of these values.) The default value is zero.

```
10 DIM BOXES%(500)
20 FOR J=2 TO 50 STEP 2
30 PENO J MOD 3 + 3
40 BOX(50,50; 50+J, 50+J)
50 NEXT J
60 SSHAPE(50,50;110,110), BOXES%()

40 GSHAPE (120,50), BOXES%()
```

When you run the program, the result is:



The original box on the left is duplicated on the right by GSHAPE.

LINEPAT

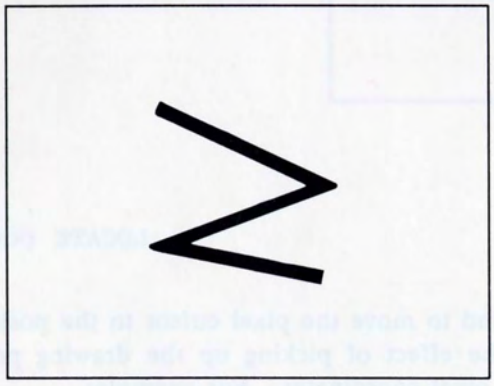
`LINEPAT <integer>`

The LINEPAT command sets the pattern for line drawing. You can specify any integer expression, but LINEPAT uses only the low–order 16 bits. LINEPAT uses the current PENA color for the "1" bits in the pattern and leaves the "0" bits alone. For closed figures that you define with the AREA and BOX commands, LINEPAT uses the current PENO color.

The default LINEPAT value is 65535. This value is simply sixteen "1" bits in binary form:

```
1111111111111111
```

If you execute a DRAW statement with the default pattern, it might look like this:



You can obtain a dot–dash line pattern with the following binary number:

```
1001110010011100
```

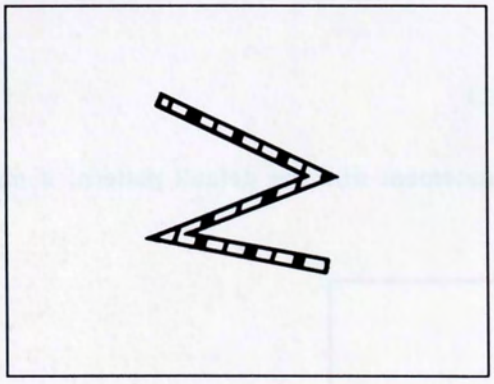
This translates into 156 x 256 (high–order byte) + 156 (low–order byte), or 40092 in decimal. You specify this pattern with either of the following commands:

```
LINEPAT 40092
```

or

```
LINEPAT  &B1001110010011100
```

Using the new LINEPAT pattern, the drawing above would look like this:



LOCATE

LOCATE (<x>,<y>)

Use the LOCATE command to move the pixel cursor to the position you specify. LOCATE has the effect of picking up the drawing point and placing it at the specified pixel coordinates. For example:

```
10 LOCATE (120,10)
20 DRAW (TO 100,100)
```

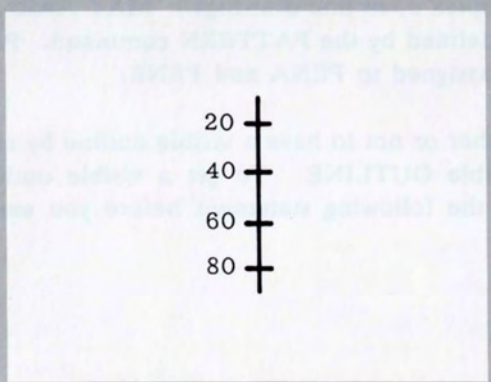
These two lines position the pixel cursor at point (120,10) and draw a line from there to point (100,100).

LOCATE is useful for applications such as labeling business graphics. For example, you can put markers at precise locations on a graph with a loop, as in the following programming example.

```
30 DRAW (23,10 TO 23,100)

50 FOR K = 20 TO 80 STEP 20
60 LOCATE (20, K)
70 DRAW (TO 25, K)
80 PRINT AT (1, (K)/8); K;
90 NEXT K
```

When you run the program, the result is:



MAT AREA

MAT AREA <integer> ,<array%>

Use the MAT AREA command to define and flood fill a closed area on the screen. (You can use the AREA command for shapes that have few enough vertices to fit on a single program line.) Each area must include at least three points. The integer you specify indicates the number of coordinate

pairs to read in the specified array. The array elements are the coordinates of the successive vertices. For example, when ABasiC encounters the statement

```
MAT AREA 3, TRI%()
```

the TRI%() array should have six elements that describe three points on the current output screen:

```
x1, y1, x2, y2, x3, y3
```

When MAT AREA executes, it automatically connects the last point specified with the first point and fills in the region. (See the MAT DRAW command for making complex *open* line drawings.) MAT AREA uses the fill pattern most recently defined by the PATTERN command. PATTERN uses the colors currently assigned to PENA and PENB.

You can also choose whether or not to have a visible outline by setting the value of the system variable OUTLINE. To get a visible outline for a particular shape, include the following statement before you execute the MAT AREA statement.

```
OUTLINE 1
```

If OUTLINE is 1, ABasiC uses PENO's current color and the current LINEPAT pattern to draw the outline. (The default LINEPAT pattern is a solid line, sixteen "1" bits).

If you specify the following:

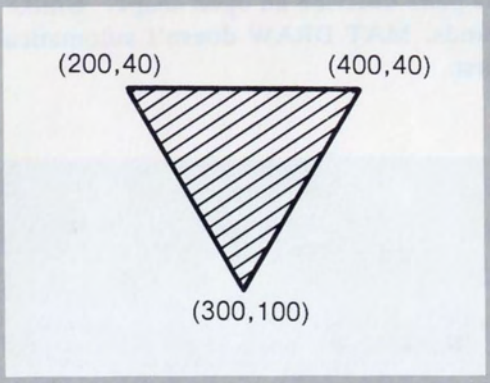
```
OUTLINE 0
```

the outline is invisible. The pattern simply fills to the border defined by the specified vertices.

```
10 DIM TRIANGLE%(12)
20 FOR I=0 TO 5
30 READ TRIANGLE%(I)
50 NEXT I
60 MAT AREA 3, TRIANGLE%()

100 DATA 200, 40, 400, 40, 300, 100
```

When you run the program, the result is:



MAT DRAW

MAT DRAW <integer>,<array%>

Use the MAT DRAW command to define and draw lines between the points you specify in the array. (You can use the DRAW command for shapes that have few enough vertices to fit on a single program line.) Each shape must include at least two points. The integer you specify indicates the number of coordinate pairs to read in the specified array. The array elements are the coordinates of the successive vertices. For example, when ABasiC encounters the statement

```
MAT DRAW 4, Z% ()
```

the Z%() array should have eight elements that describe four points on the current output screen:

```
x1, y1, x2, y2, x3, y3, x4, y4
```

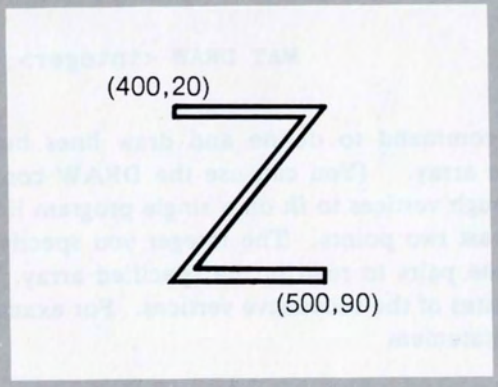
MAT DRAW uses the color currently assigned to PENA and the current line pattern (see the LINEPAT command).

Three or more coordinate pairs describe an open shape. Unlike the AREA and MAT AREA commands, MAT DRAW doesn't automatically connect the last point with the first.

```
10 DIM Z%(12)
20 FOR I=0 TO 7
30 READ Z%(I)
50 NEXT I
60 MAT DRAW 4, Z%()

100 DATA 400, 20, 500, 20, 400, 90, 500, 90
```

When you run the program, the result is:



PAINT

`PAINT (x,y) [,<mode>]`

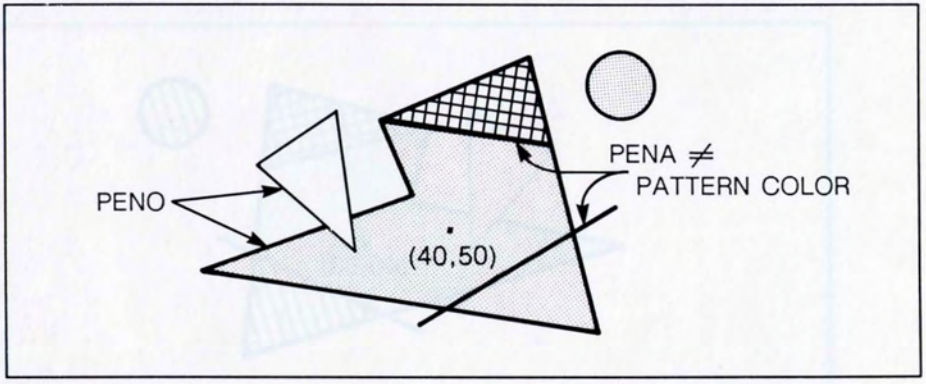
The PAINT command flood fills an area with the current pattern (see the PATTERN command). PAINT uses the current DRAWMODE and the colors currently assigned to PENA and PENB. Depending on which mode you use, PAINT can also use PENO.

PAINT uses the location (x,y) as the starting point for the fill. The command's behavior depends on which of the two modes (0 or 1) you use:

0 Fills all adjacent pixels from the starting point to the boundary defined by PENO (the outline pen color). This is the default mode.

1 Fills all adjacent pixels that are the same color as that of the drawing point (x,y), and stops if there is another color.

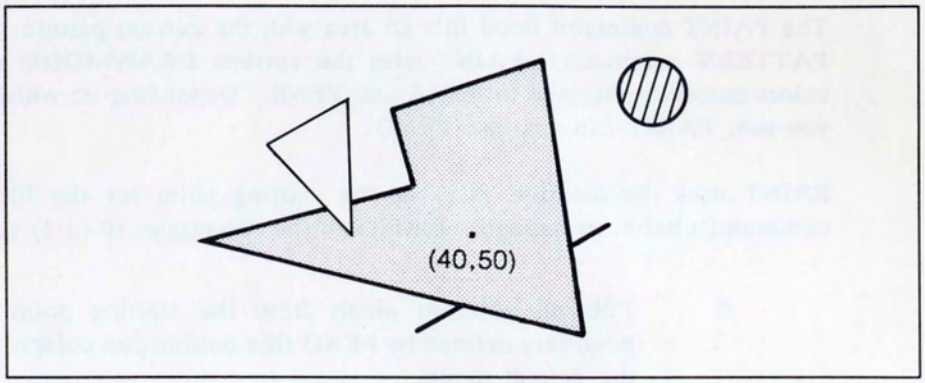
For example, assume you have the fill PATTERN currently set to sixteen "1" bits, or a solid fill, and you have created the following figure on the screen that you want to PAINT:



The statement

```
PAINT (40,50)
```

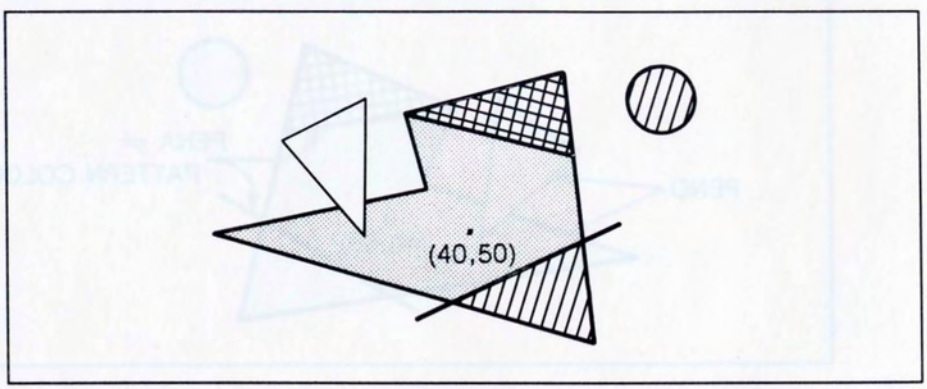
makes the drawing look like this:



The pattern stops only at borders defined by PENO. If you specify mode 1 for the same drawing, i.e.,

```
PAINT(40,50) ,1
```

it looks like this:



In the second example, the pattern stops when it encounters a different color than the color at point (40,50).

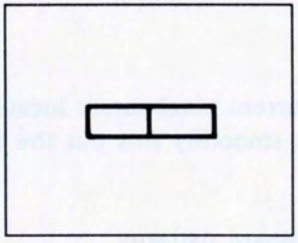
PATTERN

PATTERN <integer>, <array%>

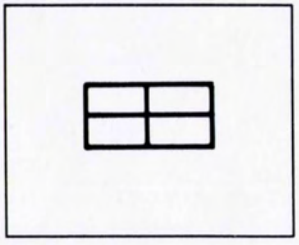
The PATTERN command lets you define the floodfill pattern used by the flood fill commands, such as AREA and PAINT. Specify the number of integers you want to be part of the pattern as the first argument. Then specify an array that contains at least that many pattern values.

The pattern always defines a rectangular area. The rectangle's size depends on the number of integers you specify and the lowest power of two they fit in, as follows:

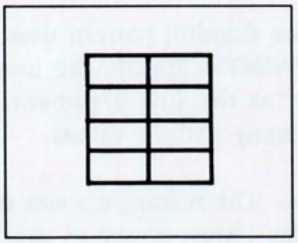
Number of Integers	Size of Rectangle
1 to 2	2 x 16–bit words (one word equals two bytes):



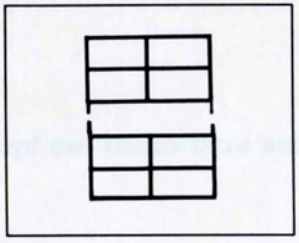
3 to 4	4 x 16–bit words:



5 to 8 8 x 16–bit words:

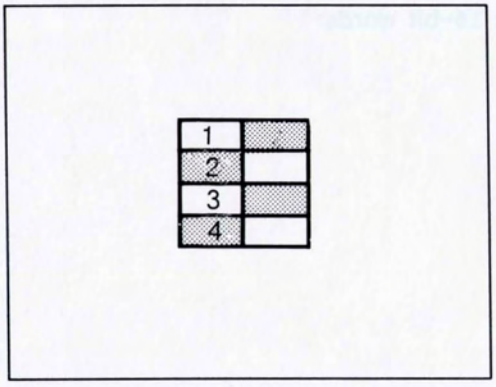


9 to 16 16 x 16–bit words:



The defined pattern begins relative to the current pixel cursor location and behaves somewhat like a "seed," in that it smoothly fills out the area or output window as far as it can go.

If you want to create the following checkerboard pattern:



specify the following PATTERN statement:

```
PATTERN    4, CHECKER%()
```

where the first four elements of array CHECKER%() are:

```
255 , 65280 , 255 , 65280
```

Binary 0000000011111111 is 255; binary 1111111100000000 is 65280.

For larger or more intricate patterns, just sketch the pattern you want, translate it into a pattern of 1's and 0's (a 16–bit binary number), and figure the decimal equivalent of the result. The decimal number is the integer you specify for the array element corresponding to that square.

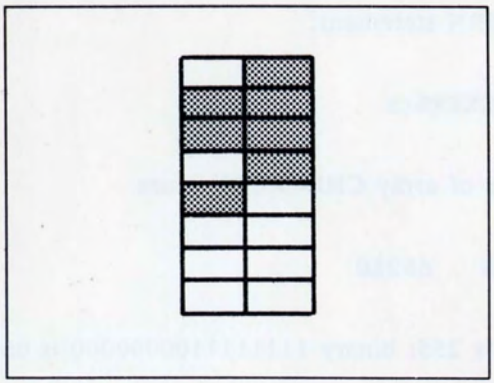
If your integer argument isn't a power of two, ABasiC fills out remaining cells in the rectangle with all "0" bits. For example, if you specify the following statement:

```
PATTERN  5,  MISC%()
```

where the first five elements of array MISC%() are:

```
255,65535,65535,255,65280
```

you describe an 8 x 16–bit pattern. ABasiC assigns binary 0 to integers 6 through 8 to complete the pattern:



PENA

PENA <integer>

Use the PENA command to specify which color register's contents are to be used as the primary drawing pen color. The primary pen is the one used for foreground color; this includes line drawing, text printing, and the "1" bits in area fills.

The argument must be an integer between 0 and 31. If you specify a value that is out of range, the lower five bits are used. (Six or more bits describe a minimum of binary 32, which is too large to refer to one of the color registers.) Note that only the first 16 color registers are used under default conditions. The upper 16 (registers 16 to 31) contain duplicate color compositons of the lower 16 values. You must use the RGB command to put different values in the upper 16 registers.

```
PENA 12
```

This statement sets the primary pen color to the current value in color register 12. The default values of the first 16 color registers are listed in the table entitled "Default Color Register Values" at the end of this subsection.

PENB

PENB <integer>

Use the PENB command to specify which color register's contents are to be used as the secondary pen color. The secondary pen is the one used for the background (the "0" bits) in area fill patterns.

The discussion of range and limits of PENA values applies to PENB as well.

PENO

PENO <integer>

Use the PENO command to set the color of an area outline to the one currently defined for the specified color register 0 to 31. The outline color is the one used to enclose areas for flood filling.

If you set the system variable OUTLINE 1, ABasiC uses the current PENO value to draw a visible outline for the shape. PENO's color is used with the following commands:

```
AREA
BOX
CIRCLE
MAT AREA
```

AREA, BOX, and MAT AREA all use the current LINEPAT pattern. CIRCLE just uses PENO's color to make a solid outline.

If you specify OUTLINE 0, ABasiC ignores the PENO color and executes the flood fill shapes with an invisible outline.

RGB

RGB <integer>, <integer-1>, <integer-2>,<integer-3>

The RGB command allows you to define the color composition in the specified register (the first integer). The color is a combination of the red (integer-1), green (integer-2), and blue (integer-3) colors. The color register you select must be between 0 and 31, and the values you give for

the respective color variables must be integers between 0 and 15. For example, to define pure green for color register 4, use the following statement:

```
RGB 4, 0,15,0
```

That is, the value of green is the maximum value of 15, while those for red and blue are zero.

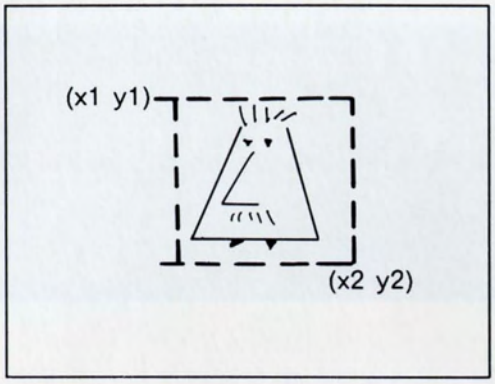
See the table at the end of this subsection entitled "Default Color Register Values" for the default values of color registers 0 through 15.

SSHAPE

SSHAPE (x1,y1 ;x2,y2) ,<array%>

The SSHAPE (Save SHAPE) command allows you to temporarily save a rectangular portion of the display. The first pair of coordinates are the pixel position of the upper left corner of the rectangle. The second coordinate pair define the rectangle's lower right corner. The integer array saves the display contents of the rectangle. (You must DIMension this array to an adequate size before using the SSHAPE command.)

You can use the ASK MOUSE command to find the coordinates of any position on the screen (or current output window). You should also use the GRAPHIC command with a nonzero value to prevent scrolling. Suppose you have a figure you want to save with SSHAPE on the screen. Just move the mouse to where you want the rectangle's upper left corner, as illustrated below:



Use the ASK MOUSE command in immediate mode, and record its coordinates (x1,y1). Move the mouse to the lower right corner of an imaginary rectangle that encloses all of the figure you want to save (x2,y2) and once more execute ASK MOUSE. Multiply the dimension differences--(x2-x1)*(y2-y1)--by 4, the display depth. Divide this number by 32 to get the number of words. The result is the number you should use to DIMension the array you're saving the rectangle in.

You may have to increase the size to which you DIMension the array if the rectangle crosses a word boundary (that is, an even multiple of 32 pixels) in one or both dimensions. Because large arrays eat up memory, keep the rectangle's size to a minimum. You'll find a programming example that uses SSHAPE with the GSHAPE command description.

Graphics Functions

PIXEL

`PIXEL (<x> , <y>)`

The PIXEL function returns the current color register number of the pixel at position (x,y).

```
10 N = PIXEL (140,25)
20 PRINT "Pixel color at 140,25 is ";N
```

Suppose the color at 140,25 is register 4. When you run the program, the result is:

```
Pixel color at 140,25 is 4
```

Default Color Register Values

Register No.	R	G	B	Description
0	6	9	15	dark blue
1	0	0	0	black
2	15	15	15	white
3	15	9	10	cherry red
4	14	3	0	fire engine red
5	15	11	0	orange
6	15	15	2	yellow
7	11	15	0	lime green
8	5	13	0	green
9	0	14	13	aqua
10	7	13	15	sky blue
11	12	0	14	purple
12	15	2	14	violet
13	15	13	11	tan
14	12	9	8	brown
15	11	11	11	gray

Speech and Sound Commands

You can produce an astonishing variety of sounds on your Amiga. You can create musical compositions in up to four voices, sound effects for games, auditory prompts for software--the possibilities are limitless. You can even connect your Amiga to a stereo to produce high-quality sound output. Be sure to read about sound production in the *Hardware Reference Manual*.

Your Amiga can also speak English text or, with a little more effort on your part, any text you wish. AmigaDOS must load the supporting routines into RAM when ABasiC needs them (they require about 30K of RAM). Remember that this leaves less room for your ABasiC program, especially for memory-hungry graphics code.

AUDIO

`AUDIO <channels%>,<integer>`

Use the AUDIO command to turn on or turn off sounds you described with a SOUND command. Specify an integer between 0 and 15 as a mask for the channel(s) you wish AUDIO to affect. For example, the binary for 15 is

```
1111
```

Thus, if you execute the following statement:

```
AUDIO 15, 1
```

you enable all channels to emit sounds.

To turn on sound in only one channel, specify the integer equivalent of the binary number that has a 1 in that channel position and a 0 in all the others. For example, to turn on channel 2 only, specify 4:

```
0100
```

Specify one of the following values as the second argument:

1	Turn on sound in the indicated channel(s).
0	Temporarily halt the sound in the indicated channel(s). Data for all defined sounds are saved and can resume with a subsequent AUDIO command using an argument of 1 (or any positive integer).
−1	Turn off the sound in the indicated channel(s). A new SOUND command must be executed to resume sound in the specified channel(s).

NARRATE

```
<variable%> = NARRATE ("string"[ ,<array%>] )
```

The NARRATE command speaks a string that is a list of *phoneme codes*. (Phonemes are the smallest units of speech, making up the syllables and words of spoken language.) The string can be English text that you have converted to phonemes with the TRANSLATE$ command. Or, you can construct your own phoneme string to make your Amiga speak however you wish – even in another language. (The TRANSLATE$ command can only convert English text.) See Appendix D for details on creating strings of phoneme codes.

In general, use TRANSLATE$ when you want your Amiga to speak a string that's generated within an executing program, such as a user's response to an INPUT command. Create your own phoneme strings when you know in advance what you want the Amiga to say, so that you can "fine tune" the voice quality and diction. With practice, you'll be able to translate words to phonemes quickly and easily.

NARRATE is actually a function. ABasiC assigns a value of zero to the specified integer variable if it finds no errors in the phoneme string. If ABasiC encounters a misspelled phoneme (one that it can't recognize), the variable contains the string position where NARRATE found the mistake.

In addition, ABasiC places the appropriate translation error code in the STATUS system variable. Appendix D also contains a listing of the error messages corresponding to each error code. You can use these two values to debug your phonemic translations. (Note that the function value that's returned is not valid if you use synchronization mode 1, described below.)

Your Amiga speaks the string in the manner you describe in the specified integer array, using the nine arguments described below. If you want to use all of the default values for these arguments, omit the array name. NARRATE automatically assigns the default values.

Below is a list of the array element arguments and their descriptions.

Argument	Element #	
pitch	(0)	Base pitch for the voice, in hertz. Specify a value between 65 and 320. The default is 110 (normal male speaking voice).
inflection	(1)	Inflection. Choose one of two values: 0 Inflections and emphasis of syllables (the default) 1 Monotone (robot–like)
rate	(2)	Speaking rate for the voice, in words per minute. Specify a value between 40 and 400. The default is 150.
voice	(3)	Speaking voice. Choose one of two values: 0 Male voice (the default) 1 Female voice

Argument	Element #	
tuning	(4)	The sampling frequency, in hertz. This parameter controls the changes in vocal quality. Specify a value in the range of 5000 (low and rumbly) to 28000 (high and squeaky). The default is 22200.
volume	(5)	Volume. Specify a value between 0 (no sound) and 64 (loudest). The default is 64.
channel	(6)	Channel assignment for voice output. Channels 0 and 3 go to the left audio output, and channels 1 and 2 go to the right audio output. Specify one of the code numbers from the table that follows the description of NARRATE. The default is code 10, which assigns any available left/right pair of channels.
mode	(7)	Synchronization mode. Specify one of the following integers: 0 Synchronous speech output. ABasiC waits for the completion of the current execution of NARRATE before processing further commands. The value that NARRATE returns (see discussion above) is valid. This is the default value.

Argument	Element #	
		1 Asynchronous speech output. ABasiC begins executing the current NARRATE statement and then immediately resumes processing subsequent commands. NARRATE always returns 0 in this mode, although it is not a valid value.
control	(8)	Narrator device control mode. This parameter tells ABasiC how to handle a second NARRATE statement while it is executing a first one. (It is meaningful only if you use asynchronous speech output--*mode* 1 in the preceding parameter--for the first NARRATE command.) Specify one of the following integers:
		0 Process normally. ABasiC finishes executing the first NARRATE statement and then executes the second one. This is the default mode.
		1 Stop speech processing. ABasiC immediately terminates the first NARRATE statement. (The second NARRATE is a dummy--the speech string and the other parameters are unused.)
		2 Override processing. ABasiC immediately interrupts the first NARRATE statement and executes the second one.

```
10 FOR J = 0 TO 8: READ HOW%(J): NEXT J
20 TEXT$ = "dhihs ihz yohr (ahmiy5gah
   per5sinul kumpyuw5ter) spiy4kihnx."
30 X% = NARRATE ( TEXT$, HOW%() )
100 Data 110, 0,250, 0, 22200, 64, 10, 0, 0
```

When you run the program, NARRATE uses the set of default arguments except for the speaking rate.

Channel Assignment Codes

Value	Channel(s)
0	0
1	1
2	2
3	3
4	0 and 1
5	0 and 2
6	3 and 1
7	3 and 2
8	either available left channel
9	either available right channel
10	either available left/right pair of channels (the default)
11	any available single channel

PERIOD

`PERIOD <integer>, <array%>`

The PERIOD command describes how a sound changes in pitch over its duration. The integer you specify indicates how many pairs of array elements for ABasiC to use before it begins repeating. PERIOD is an optional command. If you omit it, the SOUND command creates a steady pitch with the initial sampling period you specify in its argument list.

The Amiga uses *sampled* sound. The perceived pitch of the sound depends on the number of samples used to describe the waveform and the sampling

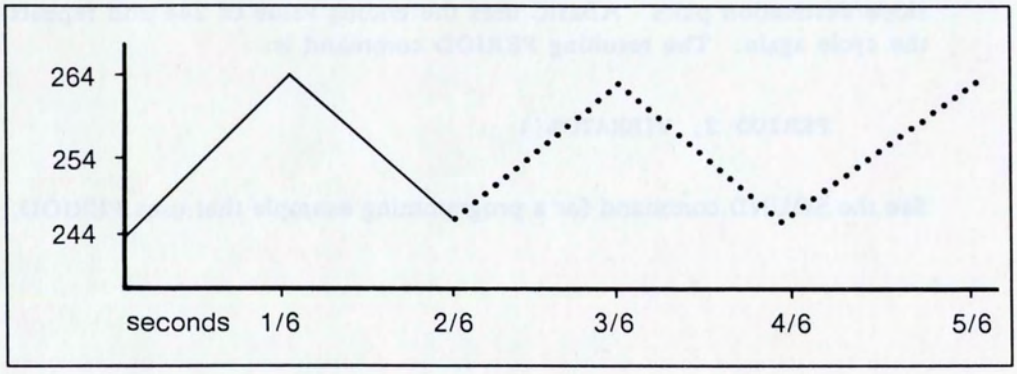
period. Suppose your waveform contains 16 samples that you play at a sampling period of 254 to produce the note A. If you cut the sampling period in half—to 127—you produce an A that is one octave higher in pitch. If other parameters remain the same, higher period values produce lower pitches, and vice versa.

The following formula describes the relationship between samples per second, time units in internal units called *color clocks* (279.365 nanoseconds), and the sampling period:

$$\text{period} = \frac{3579545}{\text{samples/cycle} * \text{cycles/second}}$$

The numerator on the right represents the number of color clocks per second. The samples are the number of elements in the waveform array defined in the WAVE command, and the cycles are the number of times the entire waveform is executed.

The integer array that you specify contains successive pairs of slope and destination values that represent changes in period values over time. You must specify an initial period value as one of the arguments for the SOUND command. The PERIOD command uses this initial period as a starting value. The following figure shows successive pairs of slope/destination values that produce a slight vibrato centered at the A above middle C:



Another formula allows you to determine the slope values in relation to time.

$$\text{slope} = \frac{1092 * \text{step-increment}}{\text{seconds}}$$

where

$$1092 = \frac{60 \text{ (60ths of a second)}}{65535 \text{ (the highest value in two bytes of memory)}}$$

and the step increment is the difference between the current period and the destination period. The slope values must be in the allowable range of integers (plus or minus 2147483648).

Suppose you call the array VIBRATO%() and specify the initial period value as 244. A rapid oscillation between 244 and 264 will sound like an A (of period 254) to the ear. Further suppose that there are three up and down vibrato cycles per second, and thus 1/6 second elapses between each extreme on the above graph. Thus, the corresponding array values for the PERIOD command to use are:

```
136500,  264, -136500,  244
```

Since the cycle starts over at that point, you only need to specify the two slope/destination pairs. ABasiC uses the ending value of 244 and repeats the cycle again. The resulting PERIOD command is:

```
PERIOD 2, VIBRATO%()
```

See the SOUND command for a programming example that uses PERIOD.

SOUND

```
<variable%> = SOUND (<channels%>,<override%>,
  <cycles%>,<init. volume%>,<init. period%>)
```

The SOUND command describes the manner of execution and makes the Amiga emit a sound in one of the audio channels. Before executing the SOUND command, define the waveform you want with the WAVE command. You can get sound out of your Amiga using *only* the SOUND and AUDIO commands. The default waveform is a sine wave, and the PERIOD and VOLUME commands are optional if you don't want the pitch or volume to change.

SOUND is actually a function. The integer variable you specify contains the channel that ABasiC assigns the sound to come out of. The first argument in the parameter list indicates which of the four channels are okay to emit the current sound, but ABasiC makes the specific assignment depending on which channels are available at the instant of execution.

With practice, you'll be able to create sophisticated sound effects and complex music in stereo with synchronized sounds coming from several channels at once. You'll find details on producing quality sound in the Audio Hardware section of the *Hardware Reference Manual*. It bears repeating that experimentation is a great teacher.

The five arguments are as follows:

Argument	Element #	
channels	(0)	A four-bit mask indicating which available channels are to produce the sound. (Channels 0 and 3 go to the left audio output, and channels 1 and 2 go to the right audio output.) The mask is the integer equivalent of a binary number between 1 and 15. For example, if you want the sound you're defining to go to either channel 1 or 2, specify 6, which is binary 0110.

Argument	Element #	
override	(1)	Channel override. Specify 0 or 1 to indicate whether you want a new sound to be able to override the sound currently executing in the specified channel(s). This feature lets you match sounds with events that occur within your program (for example, a "boom!" when two graphics objects collide).
cycles	(2)	Cycle count; the number of times to play the waveform described in the last WAVE command. Specify zero (0) for continuous sound.
volume	(3)	The initial volume of the sound. The value must be an integer between 0 (no sound) and 64 (loudest). The optional VOLUME command uses pairs of slope/destination values. The first pair begins with the initial volume you specify here.
period	(4)	The initial *period* of the sound (see the PERIOD command for a description). The pitch (how high or low a note sounds) depends on the number of times that the waveform is executed per sample period. The optional PERIOD command uses an array consisting of pairs of slope/destination values. The first pair begins with the period you specify here. The table at the end of this subsection contains the period values to use for the standard (equal-tempered) musical scale. Note that a low period value corresponds to a high frequency, and vice versa. An error results if you use a period value below 124. Period has a range of 127 to 65535, but works best from 127 to 256 or from 127 to 512 depending on the waveform. (The latter range is useful for producing lower frequencies, but at the same time introduces a whistle with the lower notes.)

ABasiC can execute sounds while it continues processing other statements in your program--including other SOUND statements. This means that you can "stack up" several sounds to come from a single channel if you want (a process called *queuing)*. Once a SOUND begins executing, it can terminate in one of three ways:

- The cycle count (specified as a positive number) is exhausted.
- Both the slope and destination values of the array used by the VOLUME command reach a value of zero (0).
- An AUDIO command disables the sound in the relevant channel(s).

```
10 DIM SINEWAV%(20), PITCH%(10), LOUD%(10)

20 FOR I=0 TO 15: READ SINEWAV%(I): NEXT I
30 FOR K=0 TO 7: READ PITCH%(K): NEXT K
40 FOR K=0 TO 5: READ LOUDNESS%(K): NEXT K

100 WAVE 8, SINEWAV%()
110 PERIOD 4, PITCH%()
120 VOLUME 3, LOUDNESS%()

130 SOUND(9, 0, -1, 32, 240)
140 AUDIO 9, 1: REM- Turn it on

150 GET A$: IF A$ <> "" THEN AUDIO 9,-1

200 DATA 0,34,64,94,100,94,64,34,0,-34,-64,-94,-100,-94,-64,-34
210 DATA 100000,250,-100000,240,-100000,230,100000,240
220 DATA 300000,32,-100000,26,-20000,12
```

TRANSLATE$ <string variable> = TRANSLATE$ ("string")

The TRANSLATE$ command converts the English string you specify into a phoneme string that becomes the value of the specified string variable. You then specify this string variable as the first argument for the NARRATE command to get your Amiga to speak it. The limit on the *resulting* string length is 255 characters.

If ABasiC expands your English string into more than 255 characters, an error results. If this happens, the STATUS variable contains the position in the string where TRANSLATE$ ran over the limit. (STATUS contains a 0 if TRANSLATE$ is able to translate the entire string.) If it does run out of room, TRANSLATE$ always stops in between words.

If your text doesn't fit in one translation string, just translate several shorter strings one after another and execute NARRATE commands to speak each one. ABasiC is fast enough to execute several commands (as part of a loop, for example) in between pairs of TRANSLATE$ and NARRATE commands without a noticeable "stutter."

```
30 A$ = TRANSLATE$ "There's no place like home."
40 X% = NARRATE (A$)
```

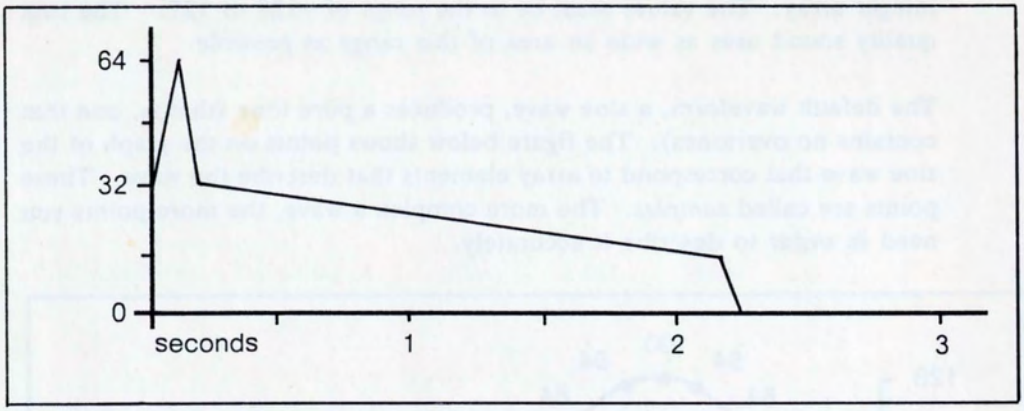
When you run the program, TRANSLATE$ converts the text to a string of phoneme codes and stores it in A$. NARRATE makes the Amiga speak the string according its default parameter list.

VOLUME

VOLUME <integer>, <array%>

The VOLUME command describes how a sound changes in volume (loudness) over its duration. The integer you specify indicates how many pairs of array elements for ABasiC to use before it begins repeating.

The array contains successive pairs of slope and destination values. The SOUND command that starts the sound contains an initial volume value. The VOLUME command uses this value as the volume from which to begin. For example, the initial value given in the SOUND command might be 32. The following figure shows how to use that value as a starting point in describing a typical sound *envelope* (the volume variation over time that distinguishes different instruments and natural sounds) for a note struck on a piano:



Suppose you call the array PIANO%(). The envelope describes a sound that increases sharply in volume (the attack), quickly fades (the initial decay), then slowly tapers to the end of the sound (the sustain) and finally ends (the release). Assume you specify an initial volume of 32 in the corresponding SOUND command. The corresponding array values for the VOLUME command to use are:

```
698880, 64,  -349440, 32,  -12600, 15,  -185640, 0
```

The resulting VOLUME command is:

```
VOLUME 4, PIANO%()
```

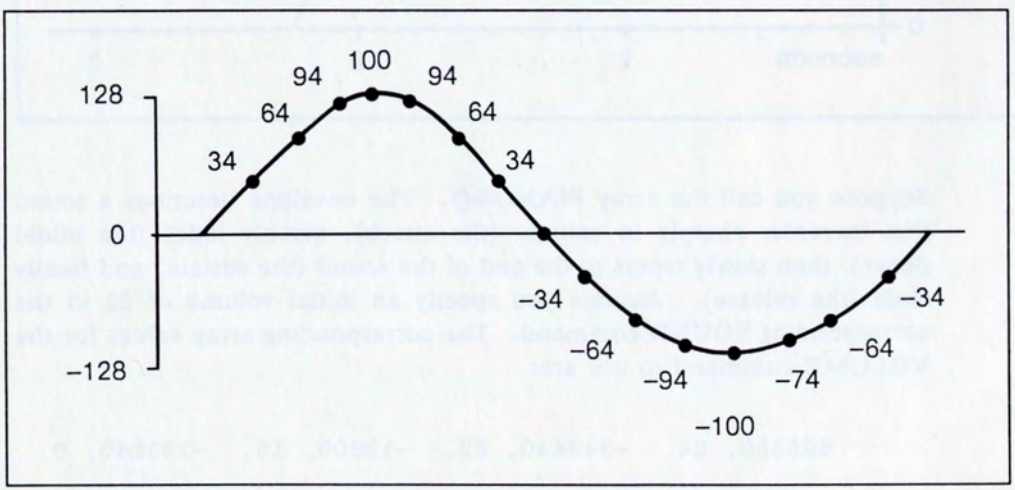
WAVE

WAVE <integer>, <array%>

The WAVE command describes the waveform of the next SOUND command to be executed. The first argument is the number of array elements that make a complete wave cycle. ABasiC will use that number of elements and then start over at the first element with each cycle repetition.

Define an integer array that describes one complete cycle of sound. See the POKE command description for instructions on storing byte values in an integer array. The values must be in the range of −128 to 127. The best quality sound uses as wide an area of this range as possible.

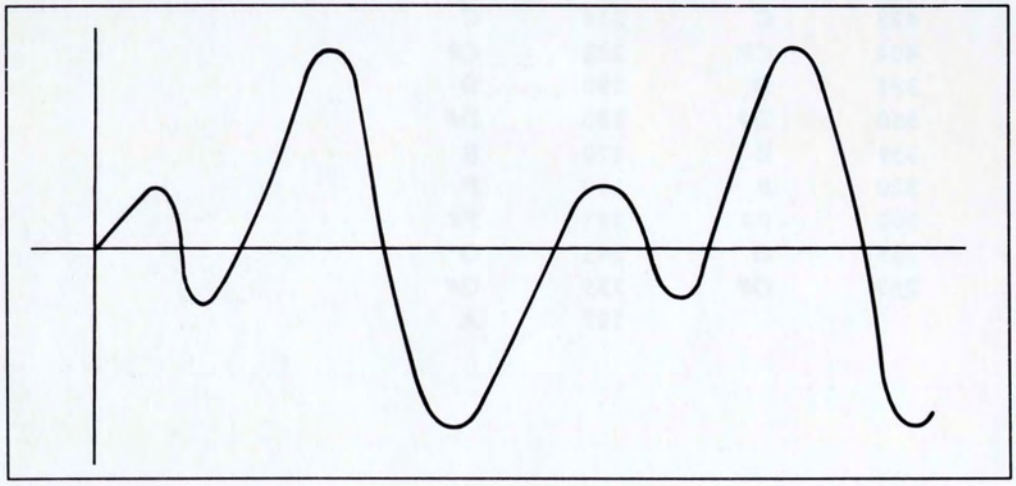
The default waveform, a sine wave, produces a pure tone (that is, one that contains no overtones). The figure below shows points on the graph of the sine wave that correspond to array elements that describe the wave. These points are called *samples*. The more complex a wave, the more points you need in order to describe it accurately.



Sample Points That Describe a Sine Wave

You should include enough samples to accurately describe the graph. To produce smooth sound, begin and end the data at or near the same cycle value.

There are a number of waveforms you can create to get different sound qualities. For example, an erratic pattern creates a sound similar to radio static, or "white noise," which has many uses. To create one kind of complex waveform, you can sum component sine waves. For example, the first overtone for the wave in the preceding figure is a tone of higher frequency than the fundamental. Thus, for every cycle of the fundamental, there are several cycles of the overtone. The sum of the two waveforms has a graph as in the figure below.



Complex Waveform as Sum of Sine Waves

Sound Functions

INPLAY

```
<variable%> = INPLAY (<integer>)
```

The INPLAY function returns an integer value that represents which of the indicated sound channels are currently producing sound. The integer returned is a 4–bit value from 0 to 15. The integer argument is also a 4-bit number that indicates which of the channels to test. For most purposes, you'll want to specify 15 to test all channels.

If INPLAY returns zero, this means there is currently no sound in any of the channels. A value of 15 means that there is currently sound coming from all four channels.

Standard Musical Scale

Period	Tone	Period	Tone
508	A	254	A
480	A#	240	A#
453	B	226	B
428	C	214	C
404	C#	202	C#
381	D	190	D
360	D#	180	D#
339	E	170	E
320	F	160	F
302	F#	151	F#
285	G	143	G
269	G#	135	G#
		127	A

File Management Commands

The contents of RAM are destroyed whenever you turn off your Amiga. You must store anything you want to save on disk in a file. ABasiC has three kinds of disk files:

- The ABasiC interpreter and the AmigaDOS operating system provided with your Amiga.
- The ABasiC programs that you write, save, run, etc.
- The data files that the ABasiC programs use.

The commands you need to create and access data files are described in "Data File Commands." The commands you need to SAVE, LOAD, RUN, etc., your programs are described in "System Commands."

The file management commands handle communications between ABasiC and the disk operating system, AmigaDOS. You can include any of these commands in an ABasiC program.

ASK WINDOW

`ASK WINDOW <width%>, <height%>`

Use the ASK WINDOW command to find out the size of the current output window. ASK WINDOW places the width (in pixels) in the first integer variable you specify and the height in the second integer variable. For example, the following lines obtain and print the size of the current output window.

```
100 ASK WINDOW wide%, high%
110 PRINT wide%, high%
```

When you run the program, the result (for a window that is 220 x 100) is:

```
220                 100
```

BLOAD

BLOAD "file-name", <address%>

Use the BLOAD command to load a binary file into memory at the specified address. You must ensure that the address is a safe area of memory—that is, one that does not overwrite any of the program or its variables, arrays, file buffers, etc. For most applications, you can declare an integer array and use the array's first cell for the file's load address.

When BLOAD executes, it checks the file for a 12-byte header that ABasiC created when you saved the file. (See the BSAVE command for a description of the header.)

You must use BLOAD for all programs that are created by the assembler or other high-level language compiler. You must load the file before you call it. You don't need to use BLOAD to load any of the resident library routines (see the *Amiga ROM Kernal Manual* for details). ABasiC has no binary file equivalent of the

```
RUN "file-name"
```

command (that is, a command that loads *and* executes a binary file).

```
10 DIM BACKGRD%(4800)
20 BLOAD GRAPH_OBJ, VARPTR(BACKGRD%(0))
...
```

When you run the program, ABasiC loads GRAPH_OBJ into array BACKGRD%.

BSAVE

BSAVE "file-name", <address%>, <length%>

Use the BSAVE command to save the specified binary file in a form that ABasiC programs can load and use. The address you specify is the location where the file begins in memory, and the length is the number of bytes in the file. Both address and length must be integers.

ABasiC automatically assigns a 12-byte header to any file you save with BSAVE. ABasiC then uses the header when it loads the file. The header format is:

4 bytes	file identification (internally generated)
4 bytes	address from which file was saved
4 bytes	length of file (in bytes)

Below is an example of BSAVE:

```
20 BSAVE GRAPH_OBJ, VARPTR(BACKGRD%(0)), 4786
```

CHAIN

CHAIN "file-name" [<line number>] [, ALL]

The CHAIN command lets a resident ABasiC program (the program currently in memory) replace itself with another ABasiC program from disk. You can also pass variables from the old program to the new before it executes.

The chained program executes as soon as it is loaded. If you specify the optional line number, execution begins at that line; otherwise, the program starts at the first executable statement. (Remember to change the line number reference if you renumber the lines in the chained program.)

The optional keyword ALL tells ABasiC to preserve all program variables for use by the incoming program. If you omit this keyword, you must use the COMMON command to specify which variables the incoming program will use. **Note: You should not use the ALL keyword if you want your program to be compatible with compiled ABasiC programs.**

```
CHAIN "CALCS3"
CHAIN "NEWPROG", 1200, ALL
```

The first example loads and executes the program "CALCS3"; it assumes that a separate COMMON statement names the variables to be preserved. The second example loads "NEWPROG," starting execution at line 1200; it passes ALL program variables to the incoming program.

CHAIN MERGE

```
CHAIN MERGE "file-name" [,<line number>]
    [,DELETE <line-number> - <line number]
```

The CHAIN MERGE command lets you CHAIN a program without completely overwriting the resident program. In other words, it handles *overlays*, which are program segments that are traded in and out of RAM by a controlling program when memory is scarce. (See also the MERGE system *command*.) CHAIN MERGE saves all variables, statements, and options.

If you execute a CHAIN MERGE command in a subroutine, it preserves the subroutine *stack* (the return address, and so forth). Optionally, you can delete a line or a range of line numbers in the resident program as you merge the two programs. You must make sure not to delete the line such a subroutine should return to.

The ALL keyword is not valid with the CHAIN MERGE command. CHAIN MERGE preserves the most recent OPTION BASE setting. (See a description in "Assignment Commands.")

Note: You shouldn't use the CHAIN MERGE command if you want your program to be compatible with compiled ABasiC programs.

```
20 CHAIN MERGE "SUBP2", 1500, DELETE 2500-
```

When you run the program, the overlay "SUBP2" merges with the resident program. The program deletes lines 2500 to the end of the old program, and execution begins at line 1500.

Errata to the Amiga *ABasiC Manual,* rev A 9.4.85

Page No.	Description

R-142 Addition of the CHDIR command description. Insert the following after the CHAIN MERGE command description:

CHDIR

```
CHDIR "file-spec"
```

Use the CHDIR (CHange DIRectory) command to change the current directory. When you enter ABasiC, execute a DIR command, or LOAD an ABasiC program using only a file name (that is, without a volume or drive number), the system automatically uses the main, or root, directory.

For example, if you want to store a set of programs under a special subdirectory, you can save, load, and run those programs in two ways: The first way is to specify the subdirectory and/or the drive number before the file name each time you perform one of the system commands that requires this information. The second way is to execute a CHDIR command to change the directory to the volume name and subdirectory name. Then simply specify the file name when you wish to load, save, or run a particular program.

```
CHDIR "APROGRAMS:MISC"
SAVE PROG42
```

The above commands change the current directory to subdirectory MISC on the disk with volume name APROGRAMS. You can perform the same job with the following command:

```
SAVE APROGRAMS:MISC/PROG42
```

You can use a similar format for the other system commands that involve a file specification, such as LOAD, REPLACE, RUN, and so forth.

The CHDIR command also affects other types of file activity, such as operations on data files. For example, instead of the following statement:

```
OPEN "O", #1, "APROGRAMS:MISC/PROG42DATA"
```

you can change the directory:

```
CHDIR "APROGRAMS:MISC"
```

and then use the following OPEN command:

```
OPEN "O", #1, "PROG42DATA"
```

Note that the DIR and SHELL commands operate on the root (that is, the highest level) directory only. They are unaffected by a CHDIR command executed in ABasiC. See the AmigaDOS Users Manual for further information on file specification.

R-169 Addition of information on file specification with certain system commands. Add the following text after the last paragraph on the introduction to the system commands:

Note: If you use a system with one disk drive, you must specify

```
DF0:<file name>
```

instead of the simple file name to instruct ABasiC to use the disk currently in the drive. Without the drive specification, "DF0:", the operating system will request that you put the WorkBench disk in the drive before it executes the system command you entered.

Similarly, if the file you want is on a particular volume or subdirectory, you must specify this information before entering the file name. For example, to RUN a disk file named "APROGRAM" from the disk with volume name "JUNK", use the following statement:

```
RUN JUNK:APROGRAM
```

If the specified volume is not in the drive, the system will request that you insert it before executing the above statement.

CMD

CMD [<file number>]

The CMD command lets you redirect program output to a special device, such as a custom window or a data file. Ordinarily, ABasiC lists programs and displays output in one window (the entire screen). You can open a custom window of the size and screen position you want within the default window (see the WINDOW command), associating the custom window with a file number. Then use the CMD command to route output to that window.

ABasiC treats all external devices (e.g., the line printer) as files. To route program output to a device other than the monitor screen, you must open the device as a file and associate a file buffer number with it. (See the special use of the OPEN command for routing output in this chapter.) The monitor screen is file number 0.

Once the window or "file" is open, use the CMD command to route program output to it. For example, the output of graphics commands or an ordinary PRINT statement's text can appear in that window or file. Follow the CMD command with the file number, which is an integer between 0 and 15. (The pound sign is optional.)

If you specify a negative number, output goes both to the screen and to the device assigned to the file buffer number that is the absolute value of the argument. For example,

```
CMD -4
```

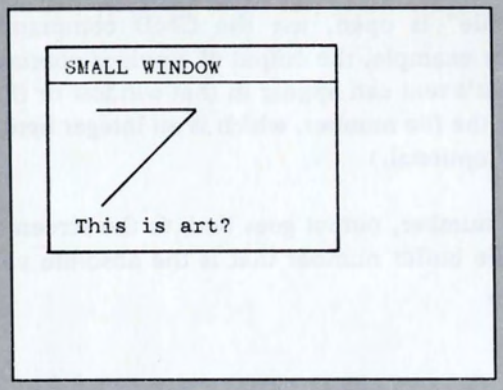
sends output to the screen as well as to the device you assigned to file buffer number 4. When you want ABasiC to resume sending output to the default window, execute another CMD command with either no argument or an argument of 0. Note that ABasiC automatically executes a CMD 0 statement when you close a window or a device.

The CMD command only sends output that is generated by an executing program or an immediate mode statement. This means you can't route system command output, such as program listings or debugging information, in this manner.

Below are sample programs for routing output to a custom window, a data file, and the line printer.

```
100 WINDOW #1, 10,10,160,100, "SMALLWINDOW"
110 CMD #1
...
150 DRAW (10,10 TO 40,10 TO 10,50)
160 PRINT AT (1, 7); "This is art?"
```

When you run the program, the result is:




```
50 OPEN "O", #3, "MYFILE"
60 CMD 3
70 PRINT AT (0,0); "This output goes to MYFILE"
```

When you run the program, the text from line 70 is stored in the sequential data file MYFILE.

```
200 OPEN "O", #2, "PRINTER"
210 CMD #2
220 PRINT "This will appear on the line printer."
...

250 CMD 0
260 PRINT "This will appear on the monitor screen."
```

When you run the program, all output between line 210 and 250 goes to the line printer. Line 250 redirects output back the monitor screen.

COMMON

COMMON <variable> [, variable...]

Use the COMMON command to name which variable(s) to pass to a chained program. (See the CHAIN command.) The CHAIN command includes an optional keyword, ALL. When you use it, all the outgoing program's variables are shared with the incoming program. If you don't use the ALL keyword, you must use COMMON to state which specific variables the two programs share.

Although the COMMON command can appear anywhere in your program, it's a good practice to place it at or near the beginning. COMMON must execute before the CHAIN statement that it affects.

```
10 COMMON AVE!, REP$, TOTAL
...
90 CHAIN "SALES", 1500
```

DEF FN

DEF FN<variable> (<variable list>) = <expression>

Use the DEF FN command to define your own functions. You must use this command to describe the function before you can execute it. The function name—the first variable you specify—can be any valid variable name. The variable list can include numeric and string variables, in any order.

In the definition of the function, you can include variables that are not in the variable list you specify. When you do so, ABasiC substitutes the existing variable value into the expression. If a variable in the definition *does* appear in the argument (variable) list, ABasiC uses its corresponding value in the statement that calls the function at run time.

To call the function, use the FN keyword followed immediately by the variable name (with no blank between them). Follow that with the argument list in parentheses, entering values in the same order as the corresponding variables in the function definition. You can't use array values with the DEF FN command.

Note that with user-defined functions the function name can't appear in the definition, causing the function to "call itself." (This process is called *recursion*.) You cause stack overflow if you attempt to define a recursive function.

```
10 Z=2
20 DEF FNTWOTIME (X,Y) = Z*X+2*Y
30 PRINT "First time:"; FNTWOTIME (3,4)
40 PRINT "Next time: "; FNTWOTIME (4,5)
```

When you run the program, the result is:

```
First time: 14
next time: 18
```

DIR (or DIRECTORY)

```
DIR [ "file-spec" ]
```

DIR, or DIRECTORY, lists all the files defined by "file spec" on the screen (or on the current output device). If you use DIR alone, ABasiC lists all the files in the current directory. Some examples of the optional file specification are as follows:

`"<dirname>"`	Lists all the files in the specified directory.
`"DFn:"`	(Where n is an integer denoting the drive number.) Lists all the files and subdirectories in the root (highest level) directory of the specified drive.

See the *AmigaDOS Users Manual* for details on further file specification options. Below are a few examples.

```
DIR

DIR "df0:"

DIR "graphdir"
```

LIBCALL

```
LIBCALL <pointer%> ,<index%>, <array%>
```

Use the LIBCALL command to call a resident library function (see the *Amiga ROM Kernal Manual* for details) or an assembly language routine. The first argument is the called routine's pointer, or the library base pointer. The second argument is the index into the library functions. LIBCALL passes the values in the first 16 cells of the integer array to registers D0–D7 and A0–A7, respectively. On return, it passes the register values back to the integer array.

You don't need to initialize the array before the call, unless the routine needs those values. However, you must dimension the array to at least 16 before LIBCALL executes.

If you are calling an assembly language routine, you must first load it into memory (see BLOAD). Then specify its starting address as the pointer and specify an index of zero (0).

The following programming example shows how to open the Exec library for the purpose of obtaining the pointer to the library function you want to call. The example uses a system variable EXECBASE, which contains the pointer to the low level library containing the OpenLibrary function.

```
10 DIM ARGS%(16)
    . . .
100 LIBNAM$ = "graphic.library" + chr$(0)
110 ARGS%(0) = 0: REM - pass to register D0
120 ARGS%(9) = VARPTR(LIBNAM$) + 1: REM - pass to reg. A1
130 LIBCALL (EXECBASE , -408 , ARGS%())
140 GRAPHPTR% = ARGS%(0)
```

OPEN

OPEN "<mode>", #<file number>, "file-name"

A special form of the OPEN command opens a device, such as the line printer, to reroute output with the CMD command. ABasiC treats external devices as special forms of files. The OPEN command described in "Data Files Commands" has several modes. Only mode "O," which opens a file for sequential output, is valid in device-related operations.

To route program output to the line printer, use:

```
OPEN "O", # <file number>, "PRINTER"
```

You must use the LIST command with the file number option to route a program listing to the line printer. The above statement is only valid for output that an executing program generates.

To route program output to a data file, use:

```
OPEN "O", #<file number>, "file-name"
```

See the CMD command for programming examples using OPEN.

PEEK PEEK (<integer expression>)
PEEK_W PEEK_W (<integer expression>)
PEEK_L PEEK_L (<integer expression>)

The PEEK functions give you access to the contents of any address in RAM. The integer expression represents the address you want to PEEK into:

PEEK returns the byte value at the specified address.

PEEK_W returns the word value (16 bits) at the address.

PEEK_L returns the long-word value (32 bits) at the address.

Clearly, the value of the specified expressions for the word and long-word forms of this function must be even.

```
PRINT PEEK_L(EXECBASE)
```

returns the address of the pointer to the Exec library.

```
PRINT PEEK(SINEWAV%(0))
```

returns the first value in the waveform array SINEWAV%().

POKE POKE <integer-1> ,<integer-2>
POKE_W POKE_W <integer-1> ,<integer-2>
POKE_L POKE_L <integer-1> ,<integer-2>

The POKE commands insert values into any address in RAM. The first integer expression represents the address where you want to insert the value. The second integer is the value you want to place at that address:

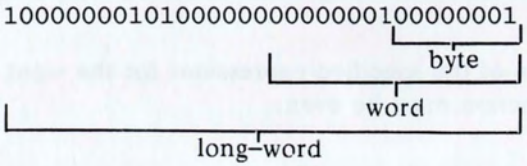
POKE inserts a byte value at the specified address.

PEEK_W inserts a word value (16 bits) at the address.

PEEK_L inserts a long-word value (32 bits) at the address.

Clearly, the value of the specified expressions for the word and long-word forms of this command must be even.

Only the low-order part of a 32-bit value is inserted into memory with POKE and POKE_W; the high-order part is ignored. For example, if you use these commands with the following 32-bit value:



ABasiC ignores the high-order part (the leftmost 16 bits). POKE_W stores 257 (the decimal equivalent of the rightmost 16 bits) in the specified address. POKE stores 1 (the binary equivalent of the rightmost eight bits) in the specified address.

One very useful application of POKE is in creating sounds for your Amiga. The WAVE command supplies a sound's waveform in the form of an integer array, yet the values are bytes. Since ABasiC assigns each integer array 4 bytes per cell, you must have a way of loading byte values into the array. The following programming example shows how to use POKE for this purpose.

```
10 DIM NOISE%(32)
. . .
50 X% = VARPTR(NOISE%(0))
60 FOR K = 0 TO 31
70 POKE X% + K, SINEWAV%(K)
80 NEXT K
```

The above program assumes you have already initialized the SINEWAV% array values.

Note: Use this group of commands with caution. You can easily destroy valuable memory contents and cause unpredictable results by inserting values at the wrong location.

SCRATCH

`SCRATCH "file-name"`

Use the SCRATCH command to delete the specified disk file. You can use SCRATCH in either immediate or program mode. You can't delete a file that is currently open; if you try, an error results.

You can use SCRATCH with any type of disk file--binary, ABasiC programs, data files, etc. Below are examples of the SCRATCH command.

```
10 SCRATCH "BKUP"

SCRATCH "MYPROG"
```

SCREEN

`SCREEN <mode%> , <depth%> , <top%>`

The SCREEN command lets you create a custom screen with a higher resolution and/or a greater number of available colors than the ABasiC default screen. An Amiga *screen* serves as a backdrop for one of more windows (see the WINDOW command). A window provides a place for line entry or program output, whereas a screen defines the appearance of that output in terms of number of colors, number of pixels and characters per line, and so on.

In ABasiC, resolution changes only affect the horizontal number of pixels. The number of scan lines (the vertical resolution) is set at 200 and can only be changed with a machine language program. Use one of the following integer values to set the screen mode:

0	Selects 320 pixels per line.
1	Selects 640 pixels per line.

Specify an integer value between 1 and 5 for the screen *depth*. The number of available colors on a screen depends on how many *planes* define it. The total possible colors are the nth power of 2 for a depth of n. Thus, a depth of 1 has a maximum of two colors--blue and white. A depth of 2 gives you a choice of four colors, 3 gives eight colors, and so on.

The depth of the default ABasiC screen is four--hence 16 colors are available when you run ABasiC. Each plane you add to the display provides more colors, but it also requires substantially more memory. Thus, you'll have to consider the memory tradeoffs in requiring a rich variety of colors for your graphics.

The last argument--top%--denotes the top of the custom screen you're defining. You can display full-color, high-resolution graphics in the bottom half of the screen while listing debug output in the top half in low-resolution white on blue. Screens automatically extend to the entire width of the monitor, so no width parameter is needed. The top of a custom screen you define extends to the bottom of the monitor.

Once your ABasiC program is finished with a high-resolution screen, execute a second SCREEN command if you wish to return the screen to default condition:

```
SCREEN 1, 4, 0
```

ABasiC recognizes only one active screen at a time.

```
50 SCREEN 2, 5, 100
60 WINDOW #1, 240, 100, 400, 80, "Sketchpad"
70 CMD #1
. . .
```

When you run the program, output that follows line 70 appears in window "Sketchpad" with 32 colors available in high-resolution.

SHELL

```
SHELL "system command string"[,#<filenumber-1>,
#<filenumber-2>]
```

The SHELL command lets you execute an AmigaDOS (operating system) command within a running ABasiC program. Certain operating system commands are not appropriate, because they affect the current line, rather than the ABasiC line in which the SHELL statement appears. For example, HELP lists the line that produced the most recent ABasiC error for editing purposes and thus has no use within an ABasiC program.

The SHELL command can process background tasks, such as printing out a report, without interrupting the other work the program has to do. Use the optional file numbers to route output from file 1 to file 2. Otherwise, the output goes to the monitor screen.

```
110 SHELL "type FEBSALES to LP"
```

When you run the program, it prints the data file named FEBSALES on the line printer. When finished, the program returns control to ABasiC.

SLEEP

```
SLEEP <numeric expression>
```

The SLEEP command suspends execution of the ABasiC program for the number of microseconds (millionths of a second) you specify in the expression. In other words, SLEEP is a built-in "wait" loop which slows down execution. The numeric expression should have an integer value.

The longest wait a single SLEEP statement can effect is 2147 seconds. Longer delays must be made with a loop.

```
20 PRINT "Hold everything!"
30 SLEEP 5*10^6
40 PRINT "Okay, go ahead."
```

When you run the program, the result is:

```
Hold everything!                    (Wait for 5 seconds)
Okay, go ahead.
```

SYSTEM

SYSTEM

Use the SYSTEM command to exit from ABasiC into the operating system, AmigaDOS. You can use SYSTEM within an executing ABasiC program, but it exits both from ABasiC and the program that was running. On the other hand, SHELL allows you to execute system commands from within a running program and returns control to the next statement in that program when the commands finish.

```
SYSTEM

35 SYSTEM
```

WINDOW

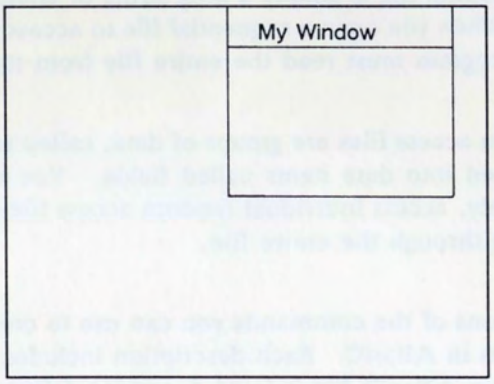
WINDOW #<file number>, <x%>, <y%>, <w%>, <h%> [, "title"]

The WINDOW command opens a custom window for your program input and output. Normally, ABasiC displays both program line listings and program output in a single window the size of the entire screen. Using WINDOW, you can display program output in a small portion of the screen. You can even direct output to different windows within the same program.

Follow the command with a file number, which can be any integer between 1 and 15. Following the file number, specify a list of integers that define the window's upper left corner coordinates (x%, y%), its width (w%), and

its height (h%). ABasiC interprets these values as pixel values, rather than character values. After the list of window dimension values, enter the window title in quotes. For example, the following WINDOW command creates the window in the figure below:

```
WINDOW #1, 100,10,140,100, "Mywindow"
```



Once you open a custom window with the WINDOW command, you can then use the CMD command followed by the window's file number to direct your program output to it. CMD also *echoes* (automatically prints keyboard input) INPUT command responses and prints INPUT prompts in the window. You must click in the custom window before typing a response to an INPUT command.

Use the SCREEN command to change the resolution and/or display depth *before* you open a window that you want to display a different resolution or depth.

```
10 SCREEN 2, 5, 0, 200
20 WINDOW #2, 10, 100, 640, 100, "bottom"
30 CMD 1
40 PRINT "This will appear in the bottom half of the
   monitor screen."
```

Data File Commands

You can store much of the information--numbers, lists of names, addresses, and so forth--that you use with your ABasiC programs as data files. You can create and use two types of data files with ABasiC:

- A sequential file is simply a long string of characters stored on disk. When you open a sequential file to access its information, your program must read the entire file from the beginning.

- Random access files are groups of data, called records, that are organized into data items called fields. You can directly, or randomly, access individual random access file records without reading through the entire file.

Below are descriptions of the commands you can use to create, read from, and add to data files in ABasiC. Each description includes the type of file to use it with. The ABasiC file-related functions follow the command descriptions.

APPEND

```
APPEND #<file number>, "file-name"
```

Use the APPEND command to open an existing sequential file for additional output (that is, to add data to it). This command is the same as mode "A" of the OPEN command. APPEND sets the file marker, the place where data will be added, at the end of the file.

The file number you specify must be one that you have not assigned to any other open file; it can be an integer between 1 and 15. This number assigns an input/output file buffer to the file; any activity with that file uses that buffer until the file is closed.

```
20 APPEND #1, "PROSPECTS"
...
50 PRINT#1, NEW$, REVISED$
```

When you run the program, the file is opened and the new string data are written at its end.

CLOSE

CLOSE [#<file number>, #<file number>...]

Use the CLOSE command to end activity and close a data file. CLOSE frees the disk buffer space assigned to the specified file number(s). The file must already be open via an OPEN statement.

You can use the command alone to CLOSE all currently open files, or you may specify one or more files by file number with the optional part of the statement. For example, if you opened two files and associated file numbers 3 and 5 with them, you can close both with the single statement:

```
CLOSE #3, #5
```

Note: The system commands NEW, RUN, CLEAR, and SYSTEM automatically close all open disk files.

FIELD

FIELD #<file number>, <field width> AS <string variable> [, <field width> AS <string variable>...]

The FIELD command allocates space for variable values in random access file buffers. All data must be in string form before you send it to the file buffer, including numbers. (See the MKD$, MKI$, and MKS$ functions for conversion.) You must use a FIELD statement to set up transfer of information between random access file buffers and program variables, although FIELD does not actually transfer the data.

The file number you specify must match the one you assigned the file when you opened it. The field width is the number of bytes to assign to the associated string variable, by means of the AS keyword. Determine each field width by the maximum number of bytes the value will require (e.g., four bytes for a double-precision variable). The FIELD statement:

```
FIELD #4, 20 AS NAME$, 30 AS POS$
```

assigns the first 20 bytes of the buffer to NAME$ and the next 30 bytes to POS$.

Don't use a variable that appears in a FIELD statement with any of the assignment commands, such as LET, =, or INPUT; if you do, the variable's pointer moves to memory that's reserved for strings or numeric variables instead of the file buffer.

The sum of the field widths can't exceed the record length you assigned when you opened the file. (See the OPEN command.) The default record length is 128 bytes, and the record limit is 4096. You can use as many FIELD statements as you wish, even if they refer to overlapping field spaces. For example, the two statements:

```
FIELD #6, 20 AS X$, 40 AS Y$, 10 AS Z$
```

and

```
FIELD #6, 70 AS Q$
```

indicate that the fields X$, Y$, and Z$ are all included in Q$. See RPUT# for a programming example that uses FIELD.

GET#

`GET #<file number>, <string variable>`

The GET# command reads a single character from a sequential data file into a file buffer. The file number you specify must match the one you assigned the file when you opened it. This command works as the GET

command does for keyboard input, except that a data file is the source of characters.

```
200 OPEN "I", #2, "Recipes"
205 FOR J = 1 TO LOF(2)
210 GET #2, A$: RECIPE$ = RECIPE$+ A$
250 NEXT J
```

(The LOF function, described in File–Related Functions, returns the number of characters currently in the file.)

INPUT#

`INPUT# <file number>, <variable> [,<variable>... ]`

The INPUT# command assigns data from a sequential disk file to program variables. The file number you specify must match the one you assigned the file when you opened it. The variables and the data types that you assign to the variables must match.

The INPUT# command can receive three types of data: any type of numeric data, quoted strings, and unquoted strings. INPUT# skips any leading blanks, tabs, or carriage returns and takes the first character that is not one of these as the first data character. Numeric data ends when the file marker reaches end–of–file, 255 characters, or it reaches a delimiter (Return, comma, space, or invalid numeric character).

INPUT# treats strings as quoted if the first non–blank character in the string is a quotation mark; quoted strings end with a second quotation mark, the end of file mark, or 255 characters.

Unquoted strings can include quotation marks and end with the Return character, a comma, or by reaching end–of–file or 255 characters. ABasiC ignores trailing blanks in unquoted strings (that is, blanks that follow the last significant character).

```
10 OPEN "I", #2, "MusicLib"
20 INPUT#2, D$, VOLUME%, CROSSREF$
```

LINE INPUT#

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

The LINE INPUT# command lets your program assign an entire logical line of data from a sequential data file to a single string variable. Like the LINE INPUT command for keyboard input, its limit is 255 characters. Input terminates with a Return character or the end–of–file marker; the next LINE INPUT# command starts where the last one left off and ends when it finds a Return, and so forth. The file number you specify must match the one you assigned the file when you opened it.

```
80 OPEN "I" #3, "PLANNING"
90 OPEN "O" #4, "PLANBKUP"
100 WHILE NOT EOF(3)
110 LINE INPUT #3, A$:PRINT#4, A$
120 WEND: CLOSE #4, #3
```

When you run the program, a new file named "PLANBKUP" is created that is an exact copy of "PLANNING."

LSET

LSET <string variable> = "string"

The LSET command transfers a string into a variable assigned to a random access file buffer without reassigning the variable. ABasiC left–justifies the string within the field width you assigned (see the FIELD command), and pads the field with blanks to the right margin if the string is shorter than the field width. If the string is longer than the field width, the characters to the far right are dropped.

You must convert numbers to strings before LSET can use them. (See the MKD$, MKS$, and MKI$ functions for details.)

```
80 FIELD #3, 15 AS A$
100 A$ = STRING$(20,"*"): INPUT "Replacement"; B$
120 PRINT A$: LSET A$ = B$: PRINT A$
```

When you run the program, the result is:

```
Replacement?12341234123412341234
***************
123412341234123
```

OPEN

```
OPEN "<mode>", #<file number>,"file-name"
     [ ,<record length>]
```

The OPEN command opens a data file for input or output. The file can be either sequential or random access. The file number you specify must be one that isn't assigned to any other currently open file; it must be an integer between 1 and 15. This number assigns an input/output file buffer to the file; any activity with that file uses that buffer until you close the file.

The file mode tells ABasiC the kind of data file it is and, if a sequential file, whether you are opening it for input or output. The file modes are:

Sequential Files

"A" Additional output to an existing sequential file; sets the file marker at the end of the file.

"I" Input from a sequential file; sets the file marker at the beginning of the file.

"O" Creates and permits output to a sequential file.

Random Access Files

"N" Creates a random access file, permits input from or output to the file.

"R" Input from or output to an existing random access file.

When you open a sequential file with mode "O" for output, ABasiC creates the file if it doesn't already exist. If the file does exist, OPEN "O" erases it and creates a new file with that name. To reopen an existing sequential file to add more data, use mode "A". (See also the APPEND command.) If you open the file with modes "A" or "I", the file must already exist or an error results.

When you open a random access file with mode "N", ABasiC creates the file if it doesn't already exist. If the file does exist, OPEN "N" erases it and creates a new file by that name. To reopen an existing random access file to read or to add more records, use mode "R". If you open the file with mode "R", the file must already exist or an error results.

The optional record length is for random access files (record length is ignored with sequential files). The default record length is 128 bytes, and the maximum is 4096 bytes. Below are a few examples of OPEN statements.

```
OPEN "N", #2, "Statements"
OPEN "R", #1, "Tax-Statements", 200
OPEN "I", #3, "Personnel"
OPEN "O", #5, "Temp"
OPEN "A", #2, "Personnel"
```

PRINT#

```
PRINT# <file number>, <expression>
[ < , | ; > <expression>...]
```

The PRINT# command works like the regular PRINT command, except it "prints" to a sequential data file instead of the monitor screen. The file number you specify must match the one you assigned the file when you opened it. You can think of each PRINT# statement as a single record. Sequential file "records" are simply character strings separated by Return characters. You can include as many expressions as you wish, as long as their total number of characters doesn't exceed 255 characters.

PRINT# writes data to the file exactly as it would print on the screen, except that the file "width" is 255. You must express exactly how you want the data to appear in the file using punctuation. For example, if you want a comma to print, use a statement such as the following:

```
280 A$ = "Choctaw": B$ = "OK"
300 PRINT#2, A$; ","; B$
```

Line 300 makes the following string print to file:

```
Choctaw, OK
```

PRINT# USING

```
PRINT# <file number> USING "string";
<expression> [ , <expression> ... ]
```

The PRINT# USING command works like the PRINT USING command, except that it formats data for disk files instead of the monitor screen. The PRINT portion of the statement is followed by "#" and the file number you specified when you opened the file. The format characters and syntax are identical to those for the PRINT USING command (see "Input/Output Commands").

RGET#

```
RGET #<file number> [ , <record number> ]
```

The RGET# command reads a record from a random access file into a file buffer. The file number you specify must match the one you assigned the file when you opened it. The optional record number lets you select the record to be read. If you omit it, the program uses record 1 or the next record after the most recent RGET# or RPUT# command executed. However, if the last operation was an RPUT#, the same record is read back from the file.

The record number must be greater than zero and less than or equal to the number of existing records in the file. (The LOF function, described in the File-Related Functions, returns the number of records currently in the file.)

```
205 FOR I = 1 TO LOF(2)
210 RGET#2, I...
250 NEXT I
```

In the above example, record number I is read into the file buffer, and the information is available to your program via the string variable(s) you set up with the FIELD statement.

RPUT#

RPUT #<file number> [, <record number>]

The RPUT command writes a record to a random access file from a file buffer. The file number you specify must match the one you assigned the file when you opened it. The optional record number lets you select the record to write. If you omit the record number, the program starts with record 1 or uses the next record after the most recent RGET or RPUT command executed. However, if the last operation was an RGET, the same record is written back to the file.

The record number must be greater than zero; it must refer to either an existing record in the file or to the next record position at the end of the file. (The LOF function, in File–related Functions, returns the number of records currently in the file.) You can't leave holes in a file; that is, you can't create record 3 if records 1 and 2 don't exist. However, you can replace record 3 if records 1 through 3 are already in the file.

Use LSET or RSET to place data into variables in the file buffer before you write them to the file with RPUT#.

```
280 OPEN "R", #2, "NOACCOUNT"
290 FIELD #2, 120 AS C$
...
305 FOR J = 1 TO LOF(2)
315 LSET  C$ = "No account activity this month"
320 RPUT#2, J
350 NEXT J
```

RSET

```
RSET <string variable> = "string"
```

The RSET command transfers a string into a variable assigned to a file buffer without reassigning the variable. RSET works like LSET, but it *right*–justifies the string within the field width you assigned (see the FIELD command), and pads the field with blanks from the left margin if the string is too short. If the string is longer than the field width, the characters to the far left are dropped. See the LSET command for a programming example.

You must convert numbers to strings before you can RSET them. See the MKD$, MKS$, and MKI$ functions for details.

WRITE#

```
WRITE# <file number>, <expression> [ ,<expression>...]
```

The WRITE# command works like WRITE, but it sends data to a sequential file instead of the monitor screen. The file number you specify must match the one you gave the file when you opened it. You must OPEN the file in mode "O" or "A" or use the APPEND command to open it.

If you plan to read the data back into memory with a series of INPUT# statements, WRITE# is preferable to PRINT#. WRITE# inserts quotes around strings and commas between items alongside the data in the file. This results in the exact form in which INPUT# uses the data.

The expressions you specify can be string or numeric data types. In addition to quotes and commas, ABasiC inserts a Return and line feed after the last item on the list.

```
20 APPEND #2, "ElectUse"
...
40 KWH = 34.275
50 K$ = "Average Kilowatt Hours per Week"
60 WRITE#2, K$, KWH
...
```

When you run the program, the following data are written in the file:

```
"Average Kilowatt Hours per Week",34.275<cr>
```

File Input/Output Functions

CVD

CVD("string")

Use the CVD function to convert an 8–byte string to a double–precision number. ABasiC stores numbers in random access files as strings, so you must convert the strings back to numbers when you read the data from the file. (See the MKD$ function to convert numbers to strings for file storage.) The value of the number doesn't through such conversions, as long as you specify the data type consistently.

If the string is shorter than the required length, it is padded to the right with binary zeros.

```
...
30 FIELD #4, 4 AS X$, 4 AS Y$, 8 AS Z$
40 RGET#4
50 I% = CVI(X$): J! = CVS(Y$)
60 K# = CVD(Z$)
...
```

CVI

CVI("string")

The CVI function works like CVD, except it converts a 4–byte string into an integer. See the description of CVD for a programming example.

CVS

CVS("string")

The CVS function works like CVD, except it converts a 4–byte string into a single–precision number. See the description of CVD for a programming example.

EOF

EOF(<file number>)

The EOF function detects the end–of–file marker when a sequential file is open for input (mode "I"). When you write to a sequential file, ABasiC notes where it ends. You can use this function to control program flow by testing for the end–of–file. EOF returns a 0 (false) if it hasn't reached the end–of–file marker and a –1 (true) if it has.

An error results if you attempt to write past the end of the file or if you use this function while the file is open for output (modes "O" or "A").

```
10 OPEN "I", #2, "Quizzes"
...
40 INPUT#2, SCORE, AV!
...
80 IF NOT EOF(2) THEN GOTO 40
90 CLOSE #2
```

LOC

LOC(<file number>)

You can use the LOC function with random access or sequential files. With random access files (mode "R" when you OPEN the file), LOC returns the current record number; that is, it returns the record most recently read with GET or written with PUT. With a sequential file (mode "I", "A", or "O" when you OPEN the file), LOC returns the number of characters that have been read or written since the file was opened.

```
30 OPEN "R", #1, "Expenses"
50 RGET#1, C%
60 IF LOC(1) > 25 THEN GOTO 90
```

LOF

LOF(<file number>)

You can use the LOF function with random access or sequential files. With a random access file, LOF returns the number of records in the file. Note that if you created the file with the current program, the value returned is

zero. LOF is useful with the RGET# or RPUT# commands, because they must refer to a valid record within the file.

With a sequential file, LOF returns the number of characters in the file. For files opened for output (mode "O" or "A"), LOF returns the same value as the LOC function; for files opened for input (mode "I"), LOF returns the total number of characters in the file.

```
20 OPEN "R", 3 "Myfile"
30 A% = LOF(3)
50 PRINT A%;" records in file"
```

MKD$

MKD$(<numeric expression>)

The MKD$ function converts a double-precision number to an 8-byte string to use in random file buffers. You must convert numbers to string form for entry into data files in ABasiC. See the CVD function to reconvert the string form to numeric.

```
40 N# = 12300.456789
50 FIELD #2 , 11 AS Z$
60 LSET Z$ = MKD$(N#)
70 RPUT #2
```

MKI$

MKI$(<numeric expression>)

The MKI$ function converts an integer to a 4-byte string.

MKS$

MKS$(<numeric expression>)

The MKS$ function converts a single-precision number to a 4-byte string.

System Commands

You can't use some ABasiC commands within a program. Typically, these *system commands*, as they are called, allow you to control an entire ABasiC program, rather than specify an action in that program. For example, the LOAD command copies an existing ABasiC program into RAM from disk. LOAD is just one of several system commands the operating system, AmigaDOS, uses to manage the files to and from disk. Other system commands perform operations on portions of the program once it is in RAM. The LIST command is an example of this type.

Several commands, such as CHAIN, relate closely to the operating system, yet they can be included as statements in an ABasiC program. CHAIN and other commands relating to files are covered in the subsection "File Management Commands."

In addition to the system commands, you can only use the EDIT command and the various debugging commands in immediate mode. Descriptions of these commands are in their own subsections.

AUTO

```
AUTO [<line number>] [ ,<integer> ]
```

Use the AUTO command to turn on automatic line numbering. Once it is on, just enter a logical line and press Return. ABasiC automatically assigns the line the next available number. Below are the different forms of this command and their effects:

AUTO	Generates line numbers in increments of ten (the default), beginning with line number 10.
AUTO n	Generates line numbers in increments of ten beginning with line number n (n must be an integer between 1 and 65529).

AUTO n,m Generates line numbers in increments of m, beginning at line n (where n is between 1 and 65529).

Press Ctrl–C to turn AUTO *off*.

CONT

CONT

The CONT command restarts the execution of a program that was halted by a STOP command or by pressing Ctrl–C. CONT restarts the program at the command that follows the point where execution stopped.

CONT has no effect if the program stopped because of an error. CONT also won't work if you edit the program in break mode. Naturally, CONT doesn't work in program mode: a program that is already executing can't be continued. CONT preserves the values of variables and arrays and leaves the data in file buffers intact.

You can use CONT to help debug a program. Insert STOPs in the program to separate program segments. Then run the program one segment at a time, using CONT to continue after each segment executes. After debugging, remove the STOPs and run the entire program.

DELETE

DELETE [-] <line number> [-] [<line number>]

Use the DELETE command to delete one or more program lines from resident (currently in RAM) ABasiC program. You must specify the line number or a range of line numbers to be deleted. DELETE uses the same conventions as the LIST command to specify line ranges, although if you use DELETE without a line number or line range, it results in an error.

EDIT

Use the EDIT command to enter the ABasiC line editor. See "Editing Commands" for a description.

HELP

The HELP command prints the program line in which an error has occurred. HELP has the same effect as typing

```
LIST .
```

(that is, LIST followed by a period). See "Debugging Commands" for details.

If no error has occurred, HELP displays the message:

```
No error line set
```

The value of ERL (the error line number system variable) is zero until an error occurs.

LIST

`LIST [#<file number>] [<line number>] [-] [<line number>]`

The LIST command displays the resident ABasiC program on the monitor screen. It displays program lines in numeric order, from the lowest to the highest number. If the program is longer than 23 lines (the default window capacity), the lines scroll off the screen one by one to make room for new lines.

Press Ctrl–D to freeze the display at any point. Press Ctrl–D a second time to continue the LISTing and scrolling. You can use LIST to display a program on–screen for editing. Once the program lines are on–screen, use the line editor to modify lines or reenter them.

The LIST command has a number of options that display different portions of a program:

`LIST <line number>`	Displays the given line.
`LIST <line number> -`	Displays all lines in the program, starting from the given line number and ending at the last line of the program.
`LIST - <line number>`	Displays all lines from the lowest line number to the given line number.
`LIST <line number> - <line number>`	(The first line number must be lower than the second.) Displays all lines from the first given line number up to and including the second line number.

To send a listing to the line printer, use the following LIST option:

```
OPEN "O",#<file number>,"printer"
LIST #<file number>
```

You must first execute an OPEN# command (see "File Management Commands"), treating the line printer as a file. Then execute a LIST command followed by a pound sign (#) and the file number you assigned. You must put a blank between the LIST command and the pound sign. For example, type the following commands to list lines 30 to 60 on the line printer:

```
OPEN "O", #2, "PRINTER"
LIST #2 30-60
```

LOAD

LOAD <file name>

Use the LOAD command to load an existing ABasiC program from disk into RAM. Before it loads the program, LOAD closes all open files and clears variables and data from a program previously in memory.

You must type the file name exactly as you saved it. See the SAVE command for instructions on saving a new program; see the REPLACE command for instructions on saving a revised version of an existing program.

```
LOAD "APROGRAM"
```

The quotes around the file name are optional.

MERGE

MERGE "file-name"

Use the MERGE command to load a program into memory without completely overwriting the resident program. (See also the CHAIN MERGE option of the CHAIN command.) MERGE saves all variables, statements, and options.

MERGE doesn't destroy any of the resident program, except where line numbers conflict, as shown by the following example.

```
40 REM - lines 40 to 60
50 REM
60 REM
SAVE "OLDPROG"

NEW
20 REM - The object of the game
30 REM - is to wipe out
40 REM - all the opponent's stones
50 REM - by surrounding them
60 REM - with your own.

MERGE "OLDPROG"
```

When you MERGE and then list the new program, the result is:

```
20 REM - The object of the game
30 REM - is to wipe out
40 REM - lines 40 to 60
50 REM
60 REM
```

If any line numbers in the MERGEd program duplicate those already in memory, MERGE replaces the old lines.

NEW

The NEW command erases an ABasiC program from RAM and clears the variables, arrays, and other ABasiC values. NEW provides a clean RAM area for a new ABasiC program. If you leave an old ABasiC program in memory while you type in a new program, parts of it might remain and mix in with your new program.

Use the NEW command with care: once it clears out an old program, the program is erased and can't be recovered unless you saved it as a disk file.

All ABasiC programs are stored in ASCII format, so they can easily be LISTed once they are in memory. If you want to protect your code from

being read by others, CHAIN the following program after your program finishes running:

```
10 END
```

This substitutes an uninteresting program into RAM and clears your program so that others can't LIST it.

RENAME

RENAME "old filename" TO "new filename"

The RENAME command lets you change the name of a disk file, including both ABasiC programs and data files. The following statement:

```
RENAME "letters2" TO "letters"
```

changes the existing file named "letters2" to "letters."

RENUMBER

RENUMBER [line number] [,increment] [,first line] [,last line]

You can change existing line numbers with the RENUMBER command. A common mistake is to number lines too close together and not leave room for new code to be added.

The RENUMBER command has a number of options:

RENUMBER	Generates new line numbers in increments of ten (the default), beginning with line number 10.
RENUMBER n	Generates new line numbers in increments of ten beginning with line number n (n must be an integer between 1 and 65529).

RENUMBER n,m	Generates new line numbers in increments of m, beginning at line n.
RENUMBER n,m,o	Generates new line numbers in increments of m, beginning at line n starting from old line number o.
RENUMBER ,m,o	Generates new line numbers from line 10 in increments of m, starting from old line number o.
RENUMBER n,m,o,p	Generates new line numbers in increments of m, beginning at line n. Numbering starts from old line number o and ends with old line number p.

Other forms are possible; just use commas to indicate whether argument indicates "n," "m," "o," or "p."

The last option is especially useful if you want to change a section of existing lines into a subroutine. For example, you can RENUMBER lines 30 through 85 to 200 through 300 (in increments of 10) with the statement:

```
RENUMBER 200,,30,85
```

Then add the following lines to the revised program:

```
30 GOSUB 200
...
190 END
...
310 RETURN
```

REPLACE

REPLACE [<filename>]
[,<line number> - <line number>]

Use the REPLACE command to store an ABasiC program that has already been saved on disk. If you don't specify the file name, REPLACE saves the resident program, replacing the original version. When you load a program, ABasiC remembers the name of the file and writes over the old version on disk.

If you specify a file name, REPLACE stores the resident program under that name, rather than using the name by which you LOADed the file. (You can use SAVE just as easily if you specify a file name that doesn't already exist.)

If you specify a line number or a range of line numbers, REPLACE stores only that portion of your program. If you leave out the line number list, the entire resident program is stored. Below are a few examples of the REPLACE command:

```
REPLACE
REPLACE "NEWVERSION"
REPLACE "NEWPROG" , 1000 -
REPLACE , -500
```

RUN

RUN [<filename>] [,<line number>]

Use the RUN command to start execution of the ABasiC program currently in RAM. If you don't specify the optional file name or line number, RUN begins execution of the resident program at the first (lowest-numbered) line.

When you specify a file name, ABasiC loads the program with that name (residing on disk) into RAM and executes it. Before the program is loaded, ABasiC clears RAM of any resident ABasiC program. Thus, you should use this form of the command with caution.

If you follow RUN with a comma and a line number, execution of the resident program begins at the specified line number. If the line number you specify isn't in the program, an error results.

Each time RUN executes, it clears RAM of all previously stored variables, arrays, user–defined functions, and other values. Only the ABasiC program itself remains. This is identical to the effect of the CLR command and gives the ABasiC program maximum amount of free RAM in which to run.

You can also RUN a disk file (that is, load and execute a program) using a starting line number. If you do this, separate the file name from the starting line number with a comma. (See the LOAD command for a description of loading ABasiC files from disk.)

Below are some examples of the RUN command:

```
RUN "PROGRAM1"
RUN "TICTACTOE", 500
RUN ,400
```

SAVE

SAVE <file name>

The SAVE command stores a new ABasiC program on disk. Once you LOAD the program and make any changes in it, you must use the REPLACE command to store the revised version. Attempting to SAVE a file that already exists on disk results in an error.

All ABasiC programs are stored in ASCII format, so you can use this command to save a program that another ABasiC program CHAINs into memory. (See the description of the CHAIN command for further information about program chaining.)

```
SAVE "NEWFILE"
```

Debugging Commands

Quite often the hardest programming errors, or *bugs,* to find are the most simple ones. ABasiC provides an extensive set of tools with which you can monitor your program for errors. ABasiC instantly rejects a *syntax* error in a new line. A syntax error is one in which the command or keyword spelling is wrong, the statement's elements are not in the right order, required punctuation is missing, and so forth.

However, there are many errors that ABasiC can't field for you—namely, logic errors. These are mistakes in the structure of the program: the manner in which you set up flow of control, the arithmetic expressions to be evaluated, and so forth.

The ABasiC debugging commands let you follow the values of specific variables, slow down program execution to a single statement at a time, and use other aids to find what's keeping your program from running properly. The debugging commands, like other system commands, can't be used as statements in an ABasiC program, except where noted.

You can press Ctrl–D to temporarily halt debugging output. The output consists mainly of a listing of certain lines and variable values. Press Ctrl–D again to continue the display of debugging output. This output typically displays a one–letter symbol denoting its type. For example, an "f" appears in front of the output of the FOLLOW command, and a "b" appears in front of each line listing of the BREAK command.

BREAK

```
BREAK [<line number> - <line number> ]
```

Use the BREAK command to step line–by–line through a running ABasiC program. If you don't specify the list of line numbers, BREAK begins with the first executable statement and halts at the conclusion of each line of code. As each line finishes executing, the line and its output (if any) are printed.

You can resume execution of the next line with the CONT command (see "System Commands" for a description). If you want to check a specific portion of the program, you can list one line number or a range of line numbers.

BREAK remains in effect, even through multiple runs of the program, until you execute an UNBREAK command to turn it off.

```
20 FOR I = 1 TO 2
30 PRINT "Give me a "
40 NEXT I
50 PRINT "Break!"

BREAK 30-
RUN
```

When you enter the lines, type BREAK 30–, and then run the program, the result is:

```
b 30 PRINT "Give me a"
Br CONT                          (enter CONT in immediate mode)
Give me a
b 40 NEXT I
Br CONT
b 30 PRINT "Give me a"
Br CONT
Give me a
b 40 NEXT I
Br CONT
b 50 PRINT "Break!"
Br CONT
Break!
Br
```

ERROR

ERROR <integer>

The ERROR command simulates an ABasiC run–time error within a program. It also sets the system variables ERR (ABasiC error code number) and ERL (line number of most recent error) as though an error had occurred. Specify the error code you want to simulate in the numeric expression.

You can simulate one of the ABasiC error codes and set an error *trap* (a portion of your code to which control goes in case of an error). If an error trap is in effect when the ERROR statement executes, program control goes to the error trap.

The ERROR statement stops the program (or statement) and prints the error code number if no error trap is set in your program or if you issue an immediate mode command with a mistake in it.

Two system variables assist you in both immediate and program mode:

ERR Holds the error code number of the last ABasiC error that occurred.

ERL Holds the line number in which the last error occurred. (If no error has occurred, the value is 0.)

For example, you can test your error trapping code by simulating an error that might occur in your program with ERROR. Error traps typically include tests for specific error codes or a range of line numbers in which the error occurred. By checking the values of ERR and ERL, you can tell whether the code works as you expect it to.

```
10 ON ERROR GOTO 100
...
45 ERROR 2
...
100 IF ERR = 2 THEN PRINT ERR$(2)
110 IF ERL > 1500 THEN RESUME 1000
120 RESUME 50
```

When you run the program, line 45 sends control to line 100. This is a handy way to test your error trapping code.

ERR$

ERR$ (<integer>)

ERR$ is a function that returns the error message that corresponds to the integer you specify. If this value is not one of the ABasiC error codes, ABasiC prints message number 0, "Undefined error."

See the ERROR command for a programming example.

FOLLOW

FOLLOW <variable> [,<variable>...]

Use the FOLLOW command to track the values of selected variables throughout a program's execution. The variables cannot be arrays. Each time a listed variable's value changes, that value is printed together with the number of the line that caused the change.

FOLLOW remains in effect until you turn it off using the UNFOLLOW command. Executing a NEW or LOAD command, which resets the variable you are tracking, also turns off FOLLOW.

```
10 X = 10
20 GOSUB 100
...
80 END
100 X = X+2: PRINT X
120 RETURN

FOLLOW X
RUN
```

When you run the program, the result is:

```
f  X! = 10 At line 10
f  X! = 12 At line 100
12
ok
UNFOLLOW
```

Notice how ABasiC performs an automatic type conversion of variable X to X!, since the default variable type is single-precision.

HELP

HELP

The HELP command displays the program line that is the current value of the system variable ERL (error line). (See the ERROR command for a description of ERL.) If no error occurred while the program ran, HELP prints the following message:

```
No error line set
```

Otherwise, HELP lists the line number where the last error occurred.

TRACE

TRACE [<line number> - <line number>]

Use TRACE to step through program execution line by line and print the specified lines as they run. TRACE doesn't interrupt program execution, as BREAK does. If you omit the list of line numbers, TRACE steps through the whole program and prints each line just before it executes.

If you include a line number or a list of line numbers, TRACE prints only the specified lines before it executes. However, the entire program runs.

The TRACE command remains in effect until you turn it off using the UNTRACE command.

```
10 FOR I = 1 TO 3
20 REM- This program does nothing very well.
30 NEXT I

TRACE 20
RUN
```

When you run the program with TRACE in effect, the result is:

```
t  20 REM- This program does nothing very well.
t  20 REM- This program does nothing very well.
t  20 REM- This program does nothing very well.
ok
UNTRACE
```

TROFF

TROFF

The TROFF (TRace OFF) command turns off the effects of the TRON command. (See the description of TRON for an example.)

TRON

TRON [<line number> - <line number>]

The TRON (TRace ON) command lets you selectively trace the line–by–line execution of a program. As each line executes, its line number and the value of all currently active variables print, enclosed in square brackets. This way, you can see how values are changing and the exact line that causes those changes.

The TRON command remains in effect until you turn it off with the TROFF command.

```
10 X = 2: Y = 3
20 FOR J = 2 TO 3
30 PRINT X*J, Y+J
40 NEXT J

TRON 30-40
RUN
```

When you run the program with TRON in effect, the result is:

```
[30] 4          6
[40] [30] 6     9
[40]
ok
TROFF
```

The line number is followed by the values of J, X, and Y, and then the line's output, if any.

UNBREAK

The UNBREAK command turns off the BREAK command. (See the description of BREAK for an example.)

UNFOLLOW

The UNFOLLOW command turns off the FOLLOW command. (See the description of FOLLOW for an example.)

UNTRACE

The UNTRACE command turns off the TRACE command. (See the description of TRACE for an example.)

Appendices

Appendix A: Quick Command Reference

Arithmetic Operators

+	Adds; concatenates strings	R–11
–	Subtracts ; negates	R–11
*	Multiplies	R–11
/	Divides	R–11
\	Converts to integers and divides	R–11
^ or **	Exponentiates	R–12

Line Editor Commands

EDIT	Starts edit mode at specified line	R–18
(down arrow)	Moves cursor down one line	R–18
(left arrow)	Moves cursor left one character	R–19
(right arrow)	Moves cursor right one character	R–19
(up arrow)	Moves cursor up one line	R–18
L	Moves cursor to start of logical line	R–19
R	Moves cursor to end of logical line	R–19
Esc	Exits insert mode	R–19
C	Replaces specified character(s)	R–19
D	Deletes specified character(s)	R–19
H	Deletes characters to cursor's right	R–19
I	Starts insert mode	R–20
K	Searches for specified character and deletes characters to its right	R–20
S	Searches for specified character and moves cursor to it	R–20
X	Moves cursor to end of logical line and enters insert mode	R–20
Z	Deletes carriage return	R–20
A	Restarts edit, ignoring all changes	R–20
E	Exits edit mode, saving all changes	R–20
Q	Exits edit mode, ignoring all changes	R–20

Operators

Relational Operators		R–21
=	Equal	R–22
<	Less than	R–22
>	Greater than	R–22
<=	Less than or equal	R–22
>=	Greater than or equal	R–22
<>	Not equal	R–22

Assignment Commands

Input/Output Commands

Program Flow Commands

Functions

Graphics Commands

ASK RGB	Fetches color composition (r,g,b) in specified color register	R–99
[ASK WINDOW]	Fetches current output window's size in width & height variables	R–100
BOX	Draws a box on screen/window	R–100
CIRCLE	Draws a circle or ellipse	R–101
DRAW	Draws line(s) between coordinates using PENA (foreground) color	R–102
DRAWMODE	Defines fore/background pen behavior for line, area, and text operations	R–103
FONT	Defines text font to use (40 or 32 column)	R–105
GSHAPE	Displays previously saved rectangle in current screen/window	R–105
LINEPAT	Defines line pattern for PENA/PENO	R–107
LOCATE	Places pixel cursor at specified position	R–108
MAT AREA	Defines array of closed area coordinate pairs for flood fill	R–109
MAT DRAW	Defines array of line draw coordinate pairs using PENA (foreground) color	R–111
PAINT	Flood fills enclosed area from specified point to confining borders	R–113
PATTERN	Defines flood fill pattern	R–115
PENA	Assigns primary (foreground) graphics pen to specified color register	R–118
PENB	Assigns secondary (background) graphics pen to specified color register	R–119
PENO	Assigns outline graphics "pen" to specified color register	R–119
RGB	Assigns specified color composition (r,g,b) to specified color register	R–119
SSHAPE	Saves specified rectangular display for graphics use	R–120

Sound and Speech Commands

File Management Commands

File Input/Output Functions

System Commands

AUTO	Automatically numbers lines on entry	R–169
CONT	Continues execution after break	R–170
DELETE	Deletes specified line(s)	R–170
[EDIT]	Turns on line editor	R–171
[HELP]	Prints error line	R–171
LIST	Lists specified line(s)	R–171
LOAD	Loads specified program	R–173
MERGE	Merges specified program with resident program	R–173
NEW	Clears memory for new program	R–174
RENAME	Renames a file	R–175
RENUMBER	Renumbers current program lines	R–174
REPLACE	Stores new version of existing program file	R–175
RUN	Executes program	R–177
SAVE	Stores a new program on disk	R–178

Debugging Commands

BREAK	Inserts breaks between lines of executing program	R–179
ERROR	Simulates an error	R–180
ERR$	Permits printing of error message	R–182
FOLLOW	Prints variable value(s) during execution	R–182
HELP	Prints error line	R–183
TRACE	Prints executing lines	R–183
TROFF	Turns off TRON	R–184
TRON	Prints variable values during execution	R–184
UNBREAK	Turns off BREAK	R–185
UNFOLLOW	Turns off FOLLOW	R–185
UNTRACE	Turns off TRACE	R–185

Appendix B: Error Codes and Messages

ABasiC Error Codes

Code Number	Message
0	Undefined error
1	(unused)
2	Syntax error
3	RETURN without GOSUB
4	Out of data
5	Illegal function call
6	Number too large
7	Out of work space
8	Undefined line number
9	Subscript out of range
10	Array was defined more than once
11	Divide by zero
12	Statement illegal in direct mode
13	Type mismatch
14	(unused)
15	String too long
16	(unused)
17	CONT works only in Break mode
18	Undefined function
19	(unused)
20	RESUME outside error trap
21	(unused)
22	Operand missing
23	Program line too long
24 – 49	(unused)
50	Field overflow
51	(unused)
52	Invalid file number
53	File not found
54	Invalid file mode
55	You cannot OPEN or SCRATCH a file already open

Code Number	Message
56	(unused)
57	Disk input/output error
58	File exists
59–61	(unused)
62	End of file
63	Invalid record number
64	Invalid file name
65	Invalid character in program file
66	Statement with no line number
67 – 98	(unused)
99	--Break--
100	(unused)
101	Program too large
102	(unused)
103	Invalid line number
104	Variable required
105	(unused)
106	Expression too complex
107	Number too large for an integer
108	Input data is not valid; restart input from first item
109	Stop
110	Subroutine calls nested too deep
111	Invalid BLOAD file
112 – 201	(unused)
202	Command not allowed
203	Line number required
204	FOR without NEXT or WHILE without WEND
205	NEXT without FOR or WEND without WHILE
206	Comma required
207	Parenthesis required
208	OPTION BASE must be 0 or 1
209	(unused)
210	Too many arguments
211–212	(unused)
213	Function defined more than once
214	Jump into loop attempted

Code Number	Message
215	Duplicate line number
216– 220	(unused)
221	System error
222	Program will not run
223	Too many FOR loops

AmigaDOS Error Codes

The STATUS system variable contains the last AmigaDos error that occurred.

Code Number	Message
102	Insufficient free store
105	Task table full
120	Argument line invalid or too long
121	File is not an object module
122	Invalid resident library during load
123	Invalid stream control block
182	Unexpected packet received; ignored by continuing
185	No selected stream for ENDREAD/ENDWRITE
186	Invalid input stream
187	Invalid output stream
188	Input stream replenish failure
189	Output stream deplete failure
195	Coroutine fault
197	Free store chain corrupted
198	Illegal QPKT
199	Illegal FREEVEC
201	No default directory specified when needed
202	Object in use
203	Object already exists

Code Number	Message
204	Directory not found
205	Object not found
206	Bad stream name
209	Packet request type unknown
210	Stream name component invalid
211	Invalid object lock
212	Object not of required type
213	Disk not validated
214	Disk not write–protected
215	Rename or alias across devices attempted
216	Directory not empty
218	Device not mounted
219	Seek failure
220	Comment too big
221	Disk full
230	Requested access not permitted
232	No more entries in directory
286	Data block sequence number error
287	Bit map corrupted (probably by user program)
289	Attempt to free a key that is already allocated
290	Attempt to allocate a key that is already allocated
293	Invalid checksum detected
295	Attempt to free an invalid lock
296	Disk (hardware) error
297	Attempt to use a key that is out of range
298	Requested action not recognized

Appendix C: ASCII Character Codes

Dec = ASCII decimal code Hex = ASCII hexadecimal code
n/a = not accessible directly from the Amiga keyboard

Dec	Hex	What to Type
0	00	Ctrl–@
1	01	Ctrl–A
2	02	Ctrl–B
3	03	Ctrl–C
4	04	Ctrl–D
5	05	Ctrl–E
6	06	Ctrl–F
7	07	Ctrl–G
8	08	Ctrl–H (or BackSpace)
9	09	Ctrl–I (or Tab)
10	0A	Ctrl–J
11	0B	Ctrl–K
12	0C	Ctrl–L
13	0D	Ctrl–M (or Return)
14	0E	Ctrl–N
15	0F	Ctrl–O
16	10	Ctrl–P
17	11	Ctrl–Q
18	12	Ctrl–R
19	13	Ctrl–S
20	14	Ctrl–T
21	15	Ctrl–U
22	16	Ctrl–V
23	17	Ctrl–W
24	18	Ctrl–X
25	19	Ctrl–Y
26	1A	Ctrl–Z
27	1B	Esc
28	1C	n/a
29	1D	n/a
30	1E	Ctrl–^

Dec	Hex	What to Type
31	1F	n/a
32	20	Space (blank)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A (or [up arrow])
66	42	B (or [down arrow])
67	43	C (or [right arrow])

Dec	Hex	What to Type
68	44	D (or [left arrow])
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	n/a
96	60	‘
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h

Dec	Hex	What to Type
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	Delete

Appendix D: Writing Phonetically for the Narrate Command

This appendix describes how to specify phonetic strings to the Narrator Speech synthesizer (through the NARRATE command). You don't need any previous experience with phonetics or with computer or foreign languages to learn this procedure. The only thing you need is a good dictionary, such as *Webster's Third International,* to look up the pronunciation of words you feel uncertain about. The beauty of writing phonetically is that you don't have to know how a word is spelled, only how it is said. Narrator lets you write down the English words that come out of your own mouth.

Narrator works on utterances at the sentence level. Even if you want to say only one word, Narrator treats it as a complete sentence. So, Narrator asks for one of two punctuation marks to appear at the end of every sentence: the period (.) or the question mark (?). If no punctuation appears at the end of a string, Narrator automatically appends a dash to it. The period, used for almost all utterances, results in a final fall in pitch at the end of the sentence.

The question mark, used only at the at the end of yes/no questions, results in a final rise in pitch. So, the question, "Do you enjoy using your Amiga?" takes a final question mark because the answer is a yes or a no. On the other hand, the question, "What is your favorite color?" doesn't take a question mark and is followed by a period. Narrator does recognize other forms of punctuation, discussed later in this appendix.

Spelling Phonetically

Utterances are usually written phonetically with an alphabet of sounds called the I.P.A. (International Phonetic Alphabet), found at the front of most good dictionaries. Since these symbols can be hard to learn and are not available on computer keyboards, the Advanced Research Projects Agency (ARPA) developed *Arpabet*, a way of representing each symbol with one or two upper case letters. To specify phonetic sounds, Narrator uses an expanded version of Arpabet.

A phonetic sound, or a phoneme, is a basic speech sound, almost a speech atom. You can break sentences into words, words into syllables, and syllables into phonemes. For example, the word "cat" has three letters and (coincidentally) three phonemes. If you look at the table of phonemes, you find that three sounds make up the word cat: K, AE, and T, written as KAET. The word "cent" translates as S, EH, N, and T, or SEHNT. Note that both words begin with c, but because the c says k in cat, the phoneme k is used. You may have also noticed that there is no C phoneme. Phonetic spelling operates on a very important concept: **Spell it like it sounds--not like it looks.**

Choosing the Right Vowel

Like letters, phonemes are divided into vowels and consonants. A vowel is a continuous sound made with the vocal cords vibrating and with air exiting the mouth (rather than the nose). All vowels use a two-letter code. A consonant is any other sound, such as those made by rushing air (like S or TH) or by interruptions in air flow by the lips and the tongue (like B or T). Consonants use a one or a two-letter code.

Written English uses the five vowels a, e, i, o, and u. On the other hand, spoken English uses more than 15 vowels, and Narrator provides for most of them. To choose a vowel properly, first listen to it. Say the word aloud, perhaps extending the desired vowel sound. Then compare the sound you are making to the vowel sounds in the example words to the right of the phoneme list. For example, the "a" in apple sounds the same as the "a" in cat and not like the "a's" in Amiga, talk, or made. Note that some of the example words in the list don't even use any of the same letters contained in the phoneme code, for example, AA as in hot.

Vowels fall into two categories: those that maintain the same sound throughout their duration and those that change their sounds. "Diphthongs" are the ones that change. You may remember being taught that vowel sounds were either long or short. Diphthongs are long vowels, but they are more complex than that. Diphthongs are the last six vowels in the table. Say the word "made" aloud very slowly. Note how the a starts out like the e in bet but ends up like the e in beet. The a is thus a diphthong in this word and "EY" represents it. Some speech synthesizers make you specify

the changing sounds in diphthongs as separate elements. Narrator assembles the diphthongal sounds for you.

Choosing the Right Consonant

Phoneticians divide consonants into many categories, but most of them are not relevant. To pick the correct consonant, you only have to pay attention to whether it is voiced or unvoiced. You make a voiced consonant with your vocal cords vibrating and you make an unvoiced one with your vocal cords silent. Written English sometimes uses the same letter combinations to represent both. Compare the sound of "th" in thin and then. Note that you make the "th" sound in thin with air rushing between the tongue and the upper teeth. In the "th" in then, the vocal cords are also vibrating. The voiced "th" phoneme is DH, the unvoiced is TH. So, the phonetic spelling of thin is THIHN whereas then is DHEHN.

A sound that is particularly subject to mistakes is voiced and unvoiced "s." The phonetic spelling is S or Z. For example, bats ends in S while suds ends in Z. Always say words aloud to find out whether the s is voiced or unvoiced.

Another sound that causes confusion is the "r" sound. The Narrator alphabet contains two r–like phonemes: R under the consonants and ER under the vowels. If the r sound is the only sound in the phoneme, use ER. Examples of words that take ER are absurd, computer, and flirt. On the other hand, if the r sound precedes or follows another vowel sound in the syllable, use R. Examples of words that take R are car, write, or craft.

Using Contractions and Special Symbols

Several of the phoneme combinations that appear in English words are created by laziness in pronunciation. For example, in the word connector, the first o is almost swallowed out of existence, so the AA phoneme is not used and the AX phoneme is used instead. Since spoken English often relaxes vowels, AX and IX phonemes occur frequently before l, m, and n.

Narrator provides a shortcut for typing these vowel combinations. Instead of spelling "personal" PERSIXNAXL, Narrator spells it PERSINUL. Anomaly becomes UNAAMULIY instead of AXNAAMAXLIY and combination changes from KAAMBIXNEYSHIXN to KAAMBINEYSHIN. To decide whether to use the AX or IX brand of vowel relaxation, try out both and see which sounds best.

Narrator uses other special symbols internally and sometimes inserts them into your input sentence or even substitutes them for part of it. If you wish, you can type some of these symbols in directly. Probably the most useful is the Q or glottal stop– an interruption of air flow in the glottis. The word Atlantic contains one between the t and the l. Narrator already knows there should be one there and saves you the trouble of typing it. However, you may stick in a Q if Narrator should somehow let a word or a word pair slip by that would have sounded better with one.

Using Stress and Intonation

Now that you've learned about telling Narrator what you want said, it's time to learn to tell it how you want it said. You use stress and intonation to alter the meaning of a sentence, to stress important words, and to specify the proper accents in words with several syllables. All this makes Narrator's output more intelligible and natural.

To specify stress and intonation, you use stress marks made up of the single digits 1–9 following a vowel phoneme code. Although stress and intonation are different things, you specify them with a single number. Among other things, stress is the elongation of a syllable. So, stress is a logical term––either the syllable is stressed or it is not. To indicate stress on a given syllable, you place a number after the vowel in the syllable. Its presence indicates that Narrator is to stress the syllable. To indicate the intonation, you assign a value to the number. Intonation here means the pitch pattern or contour of an utterance.

The higher the stress marks the higher the potential for an accent in pitch. The contour of each sentence consists of a quickly rising pitch gesture up to the first stressed syllable in the sentence, followed by a slowly declining tone throughout the sentence, and finally a quick fall to the lowest pitch on the last syllable. Additional stressed syllables cause the pitch to break its

slow declining pattern with rises and falls around each stressed syllable. Narrator uses a sophisticated procedure to generate natural pitch contours based on your marking the stressed syllables.

Using Stress Marks

You place the stress marks directly to the right of the vowel phoneme codes. For example, the stress mark on the word cat appears after the AE, so the result is KAE5T. Generally, there is no choice about the location of the number. Either the number should go after a vowel or it shouldn't. Narrator does not flag errors such as forgetting to include a stress mark or placing it after the wrong vowel. It only tells you if a stress mark is in the wrong place, such as after a consonant. Follow these rules to use stress marks correctly:

1. Place a stress mark in a *content* word, that is, one that contains some meaning. Nouns, action verbs, and adjectives are all content words. Tonsils, remove, and huge are all examples of words that tell the listener what they're talking about. On the other hand, words like but, if, is, and the are not content words. They are, however, needed for the sentence to function and so are called *function* words.

2. Always place a stress mark on the accented syllable(s) of polysyllabic words, whether content or function. A polysyllabic word has more than one syllable. "Commodore" has its stress (or accent) on the first syllable and would be spelled KAA5MAXDOHR. "Computer" is stressed on the second syllable spelled KUMPYUW5TER. If you aren't sure about which syllable gets the stress, look the word up in a dictionary.

3. If more than one syllable in a word receives a stress mark, indicate the primary and secondary stresses by marking secondary stresses with a value of only 1 or 2. For example, the word understand has its first and last syllables stressed with

stand getting primary stress and un getting secondary stress. Thus the spelling would be AH1NDERSTAE4ND.

4. Write compound words like baseball or software as one word but think of them as two words when assigning stress marks. So, spell lunchwagon as LAH5NCHWAE2GIN. Note that lunch gets a higher stress mark than wagon as the first word generally gets the primary stress.

Picking Stress Values

After you've picked the spelling and the stress mark positions correctly, it's time to decide on stress mark values. They are like parts of speech in written English. Use this table to assign stress values:

Nouns	5
Pronouns	3
Verbs	4
Adjectives	5
Adverbs	7
Quantifiers	7
Exclamations	9
Articles	0 (no stress)
Prepositions	0
Conjunctions	0
Secondary Stress	1, sometimes 2

These values only suggest a range. For example, to direct attention to a given word, you can raise its value; if you want to downplay it, lower its value. You might even want a function word to be the focus of a sentence. For example, if you assign a value of 9 to the word "to" in the sentence, "Please deliver this to Mr. Smith," you'll indicate that the letter should be delivered to Mr. Smith in person.

Using Punctuation

In addition to periods and question marks, Narrator recognizes the dash, comma, and parentheses. The comma goes where you would normally put it in a written English sentence and tells Narrator to pause with a slightly rising pitch, indicating that there is more to come. For example, you may find that you can add more commas than you use in written English to help set off clauses from each other.

The dash is like the comma except that the pitch does not rise so severely. Here's a rule of thumb: Use dashes to divide phrases and commas to divide clauses. Parentheses provide additional information to Narrator's intonation routine. Put them around noun phrases of two or more content words, for example "giant yacht." Parentheses can be particularly effective around large noun phrases like "the silliest guy I ever saw." They help provide a natural contour.

Hints for Intelligibility

Although this guide should get you off to a good start, the only sure way to proficiency is to practice. Follow these tricks to improve the intelligibility of a sentence:

1. Polysyllabic words are often more recognizable than monosyllabic ones. So say enormous instead of huge. The longer version contains information in every syllable and gives the listener three times the chance to hear it correctly.

2. Keep sentences to an optimal length. Write for speaking rather than for reading. Do not write a sentence that cannot be easily spoken in one breath. Keep sentences confined to one idea.

3. Stress new terms highly the first time they are heard.

These techniques are but a few of the ways to enhance the performance of Narrator. Undoubtedly, you'll find some of your own. Have fun.

commodore

Commodore Business Machines, Inc.
1200 Wilson Drive, West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue, Agincourt, Ontario, M1W 2K4