JBasic User's Guide

Tom Cole

Version 2.9 v(3)
May, 2011

This page intentionally left blank.

Table Of Contents

Getting Started

JBasic is BASIC, written in Java. JBasic is primarily intended as an environment for learning introductory programming concepts and skills, though it can be used to create quite complex applications if needed. It is intended to be run interactively from a *console-style* environment.

The first section of this manual will cover the general concepts behind JBasic - how to start JBasic, how to create a program, and information about how data is stored in variables, etc. This is followed by a reference section that describes each statement and function in the JBasic language, with some examples. Third, there is a section dedicated to using JBasic in a multi-user mode such as in a classroom setting. Finally, the last section describes some of the internals of JBasic and some statements and commands that are used just to support development of JBasic itself.

Notations Used In This Documentation

In this document, the typeface of the text can tell you information about what you are reading. Here are the conventions used in this document:

Notation	Meaning
Plain text	Informational or descriptive text
`KEYWORD`	A keyword in the JBasic language
`VARIABLE`	A user-supplied element of a JBasic program

Additionally, segments of sample programming code are displayed in grey text boxes. These may be examples of code, or examples of interactions with JBasic via a console command shell.

```
100 REM This is a comment in a sample program
```

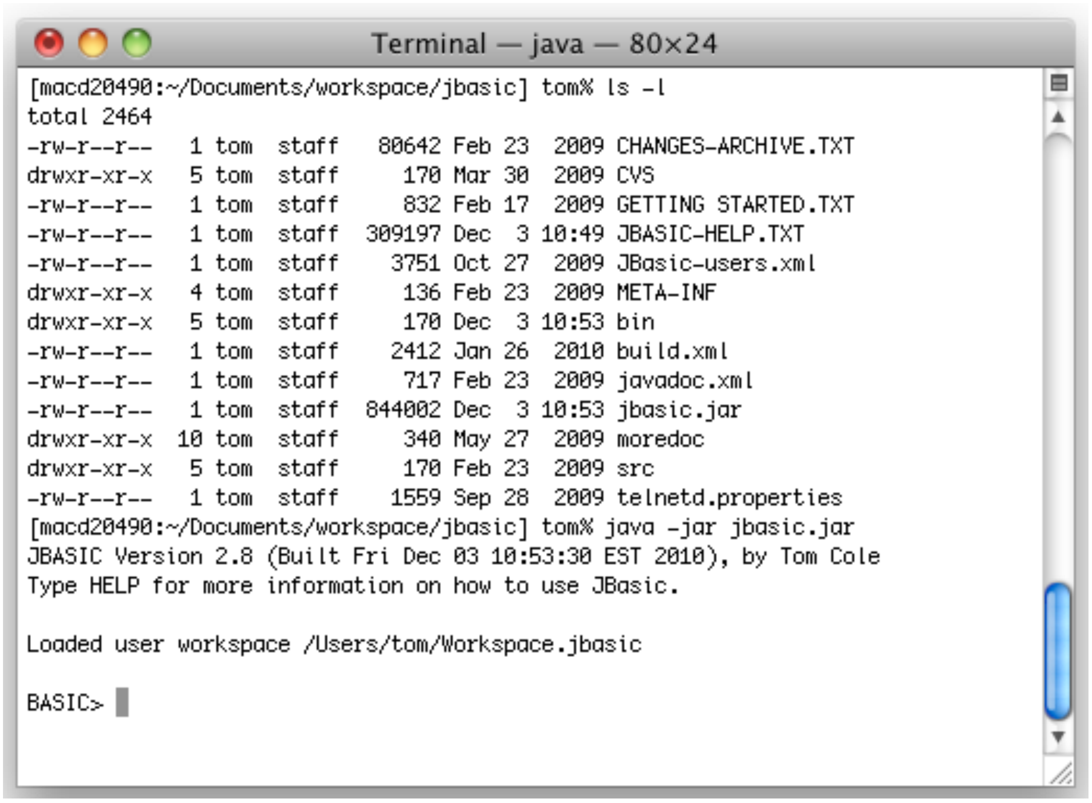
Command Line Invocation

All operations take place in the context of a JBasic "session", which is created when you use the command line shell to invoke a JBasic command prompt. The session contains both the programs you are running and the information that the programs are manipulating. When you exit JBasic, the session ends.

JBasic can be executed as a Java application using the following command, issued from within your computer's command or terminal shell:

```
java -jar <jar-location>jbasic.jar
```

In this case, *<jar-location>* is the directory path specification where the `jbasic.jar` file is located. This could be the current directory, or a shared or system location where JBasic was installed. Your system administrator may create other shortcuts or commands to assist you in running the JBasic application. By default, JBasic will print a header indicating what version you are running, and then prompt you for a command. For example,



```
[macd20490:~/Documents/workspace/jbasic] tom% ls -l
total 2464
-rw-r--r--   1 tom  staff   80642 Feb 23  2009 CHANGES-ARCHIVE.TXT
drwxr-xr-x   5 tom  staff     170 Mar 30  2009 CVS
-rw-r--r--   1 tom  staff     832 Feb 17  2009 GETTING STARTED.TXT
-rw-r--r--   1 tom  staff  309197 Dec  3 10:49 JBASIC-HELP.TXT
-rw-r--r--   1 tom  staff    3751 Oct 27  2009 JBasic-users.xml
drwxr-xr-x   4 tom  staff     136 Feb 23  2009 META-INF
drwxr-xr-x   5 tom  staff     170 Dec  3 10:53 bin
-rw-r--r--   1 tom  staff    2412 Jan 26  2010 build.xml
-rw-r--r--   1 tom  staff     717 Feb 23  2009 javadoc.xml
-rw-r--r--   1 tom  staff  844002 Dec  3 10:53 jbasic.jar
drwxr-xr-x  10 tom  staff     340 May 27  2009 moredoc
drwxr-xr-x   5 tom  staff     170 Feb 23  2009 src
-rw-r--r--   1 tom  staff    1559 Sep 28  2009 telnetd.properties
[macd20490:~/Documents/workspace/jbasic] tom% java -jar jbasic.jar
JBASIC Version 2.8 (Built Fri Dec 03 10:53:30 EST 2010), by Tom Cole
Type HELP for more information on how to use JBasic.

Loaded user workspace /Users/tom/Workspace.jbasic

BASIC> 
```

JBasic accepts commands at its console input prompt. These commands may be executed immediately, or may be compiled and stored away as programs that can be run, modified, saved, and re-loaded. You terminate use of JBasic by executing the **`QUIT`** command.

There are a variety of command line options that can follow the invocation of the JBasic jar file on the command line. These are summarized here:

Option	Description
`-exec`	The remainder of the command line is used as a JBasic command to execute. If not specified, then the main program or a command shell is used to determine what command(s) are executed by JBasic.
`-help`	Displays a description of the command line arguments.

Option	Description
`-sandbox`	Prevents JBasic programs from being able to access or change the host environment, such as being able to create files or access a command-line process. If not specified, the user's access to the host environment is controlled by the user's host attributes. This mode is typically used when JBasic is deployed as part of a web application.
`-noworkspace`	The default workspace is not loaded automatically. If not specified, JBasic attempts to load the default file "`Workspace.jbasic`" located in the user's home directory.
`-server`	JBasic starts up in multi-user mode, accepting remote connections on the default port. If not specified, JBasic runs in single user mode and only executes commands from the current user.
`-noshell`	When specified, no command prompt for JBasic commands or statements is given. After any command given on the command line or a main program has executed, JBasic terminates. If not specified, then JBasic prompts the user for commands interactively.
`-nomain`	If specified, the main program in the Library or Workspace is not executed. This is used when there is a bug in the main program that prevents JBasic from starting up normally. If not specified, then after initialization JBasic will load and run the program named "`$MAIN`" automatically.
`-nopreferences`	If specified, the preferences file "`$PREFERENCES`" is not loaded and run automatically. When not specified, JBasic attempts to run the program `$PREFERENCES` if it is found after initialization. This is used to set default user preferences.
`-nointerrupt`	If specified, the user interrupt handler is not enabled. This prevents the program from being interrupted by a Control-C or similar interrupt operation.

JBasic Programs

JBasic programs reside in a "workspace", which is a collection of programs that are available when a JBasic session is active. When you start JBasic, the default workspace is (re)loaded into memory. If you change the programs in the workspace, you can use the **`SAVE WORKSPACE`** command to preserve them so they will be available the next time you use JBasic.

JBasic has a "current program" which is the one that you are currently adding statements to or executing.

Programs must have names, identified by the first statement, which is usually a **`PROGRAM`** statement. User programs usually have line numbers, used to organize the statements in the program. You can add statements to the current program by entering a line number (any positive integer greater than zero) followed by the text of a statement. The statement is compiled and stored away.

- Use the **NEW** command to create a new program.
- Use the **OLD** command to access an existing program.
- Use the **LIST** command to see the current program.
- Use the **RUN** command to execute the current program.

Here is a simple program you can enter to get a feel for how JBasic works. Do this at the command line prompt, which is usually "BASIC>".

```
BASIC> new hello

BASIC> list
```

At this point, the program has one statement that was automatically created for you, along with a few comments. Note that in JBasic, comments are indicated by the double-slash “//” indicator rather than the keyword **REM** as in other dialects of BASIC. The output will look something like this:

```
1000     PROGRAM HELLO
1010     // Version: 1.0
1020     // Date:    Fri May 27 16:32:10 EST 2011
1030     // Author:  tom
1040     END
```

Add some more statements to the program. You can enter these in any order, since they are stored in the program in line-number order. Note that the first line has the same number as an existing line in the program; the new line will replace the old one in the program. Verify this using the **LIST** command.

```
BASIC> 1040 print "Hello, world"
BASIC> 1050 return
BASIC> list
```

Now let's run the stored program we have created.

```
BASIC> run
Hello, world
BASIC>
```

You can type **SAVE WORKSPACE** now to store this workspace (which contains the program HELLO that you just wrote). The next time you run JBasic, it will reload the stored workspace and the HELLO program will be available for your use.

You can get a list of the programs that are available in your workspace by typing **SHOW PROGRAMS**. You will see any programs you have created, plus a few that are built in to JBasic. Program names with a "*" after the name are ones that you have created or modified yourself.

You can put more than one statement on a line, with some limitations. For example, consider the following code fragment:

```
      100  X = TRUE : Y = "TOM"
      110  IF X THEN PRINT Y : GOTO DONE
      120  IF Y = "SUE" THEN X = FALSE : Y = "DEB" ELSE X = TRUE
```

The first statement shows two assignment statements performed on the same line of code. The colon character (":") separates the statements on the line. When line 100 is executed, both assignment operations will be performed. When more than one statement is on a line, this is called a "compound statement"

The second demonstrates the use of a compound statement in an **IF** statement's clause. If the value of X is a "true" value, then both the **PRINT** and **GOTO** statements will be executed. The third line shows that even in an **IF-THEN-ELSE** statement, the **THEN** and **ELSE** clauses may contain compound statements. See the reference information on the **IF** statement later in this document for more details.

About Line Numbers

JBasic programs that are typed in at the console always have line numbers. These line numbers are used to determine what order the statements are stored in the active program. Line numbers are all integer values greater than zero. Lower line numbers are stored and executed before higher line numbers.

Before you can enter or edit a program, you must make it the "current program." This is done with the **OLD** or **NEW** commands.

You enter programs statements just by typing the line number followed by the text of one or more statements (separated by a colon ":" character). For example,

```
> new hello
> 1040 print "Hello, world"
> 1050 print "Bye for now."
```

Note that these two statements could be typed in any order, the line numbers guarantee that the statements are stored in the correct order in the program itself.

When you use **NEW** to create a new program, JBasic creates a program with a few lines of text in it already. For example, the above operation would result in a complete program that looks similar to this:

```
 BASIC> list
    1000              PROGRAM HELLO
    1010              // Version: 1.0
    1020              // Date:    Fri May 27 16:32:10 EST 2011
    1030              // Author:  tom
    1040              PRINT "Hello, world"
    1050              PRINT "Bye for now."
```

JBasic automatically created lines 1000-1040 for you when you issued the **NEW** command. By convention, the use starts entering programs starting at line 1000, and incrementing by 10 to make sure there is room for additional statements. However, if you run out of room between two statements to type in new program statements, you can use the **RENUMBER** command, which renumbers the statements to make additional room for new statements. For example, given the above program, we could use the **RENUMBER** command just to make the program slightly more readable, staring at line number 100:

```
BASIC> renumber 100
BASIC> list
      100             PROGRAM HELLO
      110             // Version: 1.0
      120             // Date:    Fri May 27 16:32:10 EST 2011
      130             // Author:  tom
      140             PRINT "Hello, world"
      150             PRINT "Bye for now."
```

You can replace a line in a program just by typing the line number and a new statement to be stored in the same location in the program:

```
BASIC> 150 print "Au revoir!"
```

This replaces the line in the program with line number 150 with the new statement text. You can delete a line in a program by typing the line number without putting any text after it. To delete line 150 entirely, just type:

```
BASIC> 150
BASIC> list
      100             PROGRAM HELLO
      110             // Version: 1.0
      120             // Date:    Fri May 27 16:32:10 EST 2011
      130             // Author:  tom
      140             PRINT "Hello, world"
```

You can use the **DELETE** command to delete all the lines in a program, or a range of lines.

```
BASIC> DELETE 110-130
```

In the sample program we're working on, this would delete the comment lines that were automatically put in the program by the **NEW** command.

For compatibility with older versions of BASIC, line numbers can be used to control the “flow of execution” of your JBasic program. For example, consider the following code:

```
      100   LET X = 1
      110   IF X THEN 200
      120   PRINT “BAD”
      130   END
      200   PRINT “GOOD”
      210   END
```

This sample uses a line number as the target of a conditional branch. See the reference material describing the **IF** and **GOTO** statements later in this document for more information about flow of control and conditional branches.

If you renumber a program that uses line numbers to indicate where a program is to execute, these numbers will be changed automatically by the **RENUMBER** command so that they continue to jump to the new location. For example, if the above program were renumbered starting at 1000, the resulting program would look like the following:

```
      1000  LET X = 1
      1010  IF X THEN 1040
      1020  PRINT “BAD”
      1030  END
      1040  PRINT “GOOD”
      1050  END
```

Because line numbers are not inherently meaningful, and because they are changed when **RENUMBER** commands are given, it is suggested that you *do not* use line numbers in **GOTO** and related statement in your programs if you do not need them. See the documentation for each statement for examples of how to use text labels instead of line numbers.

Workspaces

A workspace is a collection of programs that are all loaded into the running JBasic application, and can be executed with **RUN**, **CALL**, or other statements that run stored programs, verbs (programs that act line language statements), or functions.

All the programs in a workspace are available at the same time; you don't need to specifically load and save individual programs. Any changes you make to a program (such as discussed in the section on line numbers) are immediately reflected in the program currently in memory.

A workspace can be saved to disk. This means that all the programs in the workspace that you have written are stored into a single file on the disk. The default name for this file is "Workspace.jbasic" in your home directory. (If you are running a multi-user version of JBasic, the default name will be different and will include your username.) You save the current programs to disk using the **SAVE WORKSPACE** command.

When JBasic starts up, it will load the workspace file if it is found, so the programs you previously saved are available for your use. You can use the **LOAD** command to load additional programs from saved workspaces or text files, which are added to the current workspace.

When you issue a **SHOW PROGRAMS**, **SHOW VERBS**, or **SHOW FUNCTIONS** command, you see a list of all the programs of the requested type. If the program is one that you created (as opposed to being built-in to JBasic), it will have an *asterisk* ("*") character next to the name. If you have changed the program since the last time you started JBasic or saved the workspace, it will have a *sharp* ("#") character, telling you that you will need to **SAVE WORKSPACE** to keep the changes you made.

Both the **SAVE WORKSPACE** and **LOAD** commands allow you to specify a different workspace name than the default, if you wish to create more than one workspace. You might do this to maintain separate versions of your program, or to group programs by projects or other related activity.

Note that you can save individual programs (similar to the way other BASIC dialects work) by using the **SAVE** command, which is different than **SAVE WORKSPACE** and saves only the current program to a text file. The **LOAD** command can be used to read in either individual programs or workspaces interchangeably. If the filename given in a **SAVE WORKSPACE**, **SAVE**, or **LOAD** command does not have an extension, then ".jbasic" is assumed.

Constants

JBasic supports a wide variety of data types. Most are referred to as "scalar" types, which means they contain a single value of a given type, such as an integer or a string. Some data types are "compound" and are used to hold things like lists of values, and are discussed later. This section describes the most fundamental scalar data types and how to enter values for each type.

The table below indicates the basic data types supported by JBasic:

Type	Example	Description
BOOLEAN	`true, false`	A value that can contain only the values true or false.
INTEGER	`55, -108`	An integer value with an optional sign, but no fractional component.

Type	Example	Description
DOUBLE	`-3.8, 1.2E-8`	A floating point value that can have a whole number and a fractional component. The number can be expressed in exponential notation as well, with a power-of-10 multiplier as shown in the second example, which is the same as 0.0000000012. Also note that the special value of a dot (“.”) represents an arithmetic missing value, or “Not a Number”.
STRING	`“Score:\t103”`	A string containing zero or more characters. The string can contain control characters when “escaped” with the back-slash character. The example string has a tab character between the colon and the digit “1”.

As shown in the example for a string, a string constant can include representations of characters such as the `\t` for a tab character. The table below shows the characters that can be represented in a string this way:

Value	Name	Example
`\t`	tab	`“The total is:\t103”`
`\”`	quote	`“The word \”foo\” is what you seek.”`
`\n`	newline	`“First line\nsecond line”`
`\\`	backslash	`“The \\ is used for control characters.`

Variables

Nearly all programming languages have the concept of variables, which are ways of identifying a storage location in the computer’s memory that can contain data (numbers or text). The variable can be set to a value, and the variable’s value can be used in a statement or expression by naming the variable.

In JBasic, variables must be named with an identifier, which can be made up from letters, numbers, or the "_" and "$" characters. The first character cannot be a number. Variable names are not case-sensitive; there is no difference between the name "`x`" and the name "`X`".

Here are examples of valid variable names:

```
X                       SYS$PROGRAMS

PLAN9                   MY_DATA
```

A variable can hold a single value (referred to as a "scalar" value), or it may hold a list of values in the form of an array or record. Scalar values can be numbers, strings, or true/false values. An array or record is a special variable that contains a list of values. See the sections for *arrays* or *records* later in this section for more information about these data types.

JBasic handles the idea of a variable’s data type in one of two different ways. The default mode is called “dynamic types”. This means that a variable can contain data of different types over time. For example, a variable `ID` might hold an integer value like 1105 at the start of a program, but a later operation might store a string value like “user15” in the same variable. If your program needs to know what kind of value is stored in a variable, the **`TYPE()`** function will return the type name of any variable or expression.

The second mode is called “static types.” In this mode, a variable can only hold a specific type of data, usually based on the name of the variable. The last character (the “suffix”) of the variable name defines its default type as shown in the following table.

Suffix	Example	Type Information
$	`NAME$`	Character string data
#	`COUNT#`	Integer data
!	`DONE!`	Boolean data
A-Z	`SALARY`	Floating point data

For example, the variable `ID$` would be a string variable because it ends in a dollar-sign (“$”) character. If a program stores an integer in the variable `ID$`, it would first be converted to a string. Places where a variable can only be numeric (such as the index of a **`FOR...NEXT`** loop) could not use a variable named `ID$`.

Static types are usually defined a program-by-program basis. By default, dynamic typing is used, but some programs that were originally written for other dialects of BASIC may require static types. You specify the way that variables are handled in each program by using the optional **`DEFINE(STATIC_TYPES)`** clause in the **`PROGRAM`**, **`FUNCTION`**, or **`VERB`** statement.

You can also change the default mode for all new programs by using the **`SET STATIC_TYPES`** command. You can restore the default with **`SET DYNAMIC_TYPES`**.

NOTE

The remaining examples in this guide assume the default setting of **`DYNAMIC_TYPES`**.

In many cases, you can print, assign, input, etc. a variable without concern for what type of data is stored in the variable.. You cannot perform all mathematical operations on all data types; it makes no sense to divide a string by number, for example. However, JBasic will automatically convert data types if possible, so the expression "10"/5 results in the number 2.

Expressions

Expressions refer to sequences of JBasic code that are evaluated (calculated) to derive a result. Expressions most often are performed on numeric values, but some involve string values as well. In general, expressions can appear anywhere that a value must be calculated and used, but are not used to describe where to store information.

```
LET X = 3*X+5
```

In JBasic, the above statement is correct and valid, because the expression `3*X+5` is being used to calculate what value is to be stored in the variable `X`.

```
LET X+3 = Y
```

However, the statement above is not allowed because the **`LET`** statement requires that the left side of the "=" be a *variable reference* and not an *expression*. The **`LET, INPUT, READ`**, and **`FOR`** statements all have this limitation on operations that store into a variable. An expression can be used anywhere else in the JBasic language where a numeric or string value is required.

Numeric expressions are those whose terms are all numeric, or will be converted to numbers automatically. The operators that are used to perform calculations in a numeric expression are:

Operator	Function
+	Add two numbers together
-	Subtract two numbers
*	Multiply two numbers
/	Divide two numbers
^	Raise first number to power of second number
%	Remainder (modulo) of dividing two integer values

For example, consider the following simple segment of JBasic code:

```
LET X = 3.5
LET Y = 2
PRINT X*Y
```

The result is calculated by multiplying the value of X (3.5) by the value of Y (2), and the result of the expression is output using the **`PRINT`** statement. The output would be the value 7.

Expressions follow an "order of precedence" just as they do in conventional mathematical operations. Multiplication and division operations are done first, followed by addition and subtraction.

```
PRINT 3 + Y * X
```

The above example will print the value 10, because the multiplication of `X*Y` is done before the value 3 is added back in. You can use parenthesis to group things together to control the order of evaluation.

```
PRINT (3+Y) * X
```

This will print the value 17.5, since the addition of Y and 3 will happen first, followed by the multiplication by `X`. You can use more than one set of parenthesis, and they can be "nested" inside each other to control the evaluation of complex expressions.

```
PRINT ((1+3)*(5-2))/2
```

This results in the value 6 being printed. The inner parenthesis expressions are calculated first, then the next-outermost and so on. The above expression is mathematically the same as the following examples.

```
PRINT (4*3)/2

 which is the same as typing

PRINT 12/2
```

The above examples all use numeric values. But you can express some kinds of operations using string values as well. The string operators are shown in the following table.

Operator	Function
`||`	Concatenation, which appends one string to another.
`-`	Remove a string from another string, essentially deleting a portion of a string.
`*`	Repeat a string value a given number of times, similar to the **`REPEAT()`** function.

For example, the statement

```
print "Tom" || "Cole"
```

will print out the string "`TomCole`". Note that there is no space between the items, they are concatenated directly together. Here is an example of using subtraction to delete a string:

```
print "Tom Cole" - "m C"
```

This prints the value "`Toole`", since the part that was subtracted has been deleted from the string. If you attempt to delete a string that is not there (`"Tom" - "Z"`) then the result is the first string without any changes.

You can use the leading minus sign "-" character to indicate a negative value of an expression or a constant value. For example,

```
X = 33
Y = -X
Z = -"Tom"
```

In this example, the value of `Y` will be the negative number -33. You could express the constant directly in the program as -33 as well. Additionally, you can negate a string (or have a negative string constant). In this example, `Z` is equal to "moT" which is the string with the characters in reverse order.

You can use the multiplication operator to repeat a string by multiplying it by an integer value.

```
PRINT "-"*80
```

This results in a line of eighty dashes being printed to the output. This only works if the second value is an integer; attempts to use other data types results in all values being converted to numbers and the expression being evaluated as a standard arithmetic equation rather than as a shortcut for the **`REPEAT()`** function.

For comparison operations there are a set of relational operators that compare two values and result in a **`true`** or **`false`** value.

Operator	Function
=	True if the left and right sides are equal. For scalar values, they must be the same value, after any required type coercion occurs. For record values they must have identical member names and values. For arrays, they must be identical in length and values.
<>	The inverse of the "=" operator, this is true if the values are not equal.

Operator	Function
<	True if the left side is less than the right side. This is is not valid for missing (NaN) scalar values.
<=	True if the left side is less than or equal to the right side. Not valid for NaNs.
>	True if the left side is greater than the right side. Not valid for NaNs.
>=	True if the left side is greater than or equal to the right side. Not valid for NaNs.

You can combine the relational expressions with Boolean logical operators to express ideas like "if the value of X is greater than or equal to 100 and the value of Y is not zero then…" This would be expressed as the following:

```
IF X >= 100 AND Y <> 0 THEN ...
```

Here is a table describing the boolean operations:

Operator	Meaning
`AND`	Both the left and right sides of the operator must be **`true`** for the entire expression to be **`true`**. If either or both side is **`false`**, then the expression is **`false`**.
`OR`	Either the left and right sides of the operator must be **`true`** for the entire expression to be **`true`**. If both sides are **`false`**, then the expression is **`false`**.
`NOT`	This is a "unary" operator that goes before an expression. It inverts the result of the expression, such that if what follows it is **`true`** then the result is **`false`**, and vice versa.

Increment and Decrement Operators

An expression can contain a reference to a simple numeric variable (an INTEGER or DOUBLE) that also changes the value of the variable, using increment and decrement operators. For example,

```
X = 33
Y = ++X * 2
```

In this example, the variable **X** will be incremented before it is used in the expression. The result of the assignment statement is to set the variable **X** to the value 34, and the variable **Y** to the value 68. The increment operation occurs before the value is used in the expression. You can use the decrement operator ("**--**") instead of the increment operator ("**++**") to subtract one from the value before using it in the expression. This form is referred to as a pre-increment or pre-decrement operation because the addition or subtraction of 1 is performed before the variable is used in the remainder of the expression.

You can also cause the value to be incremented or decremented after it is used in the expression. For example,

```
X = 10
Y = X-- / 2
```

The result is that the variable **X** will contain the value 9, and the variable **Y** will contain the value 5. This is because the current value of the variable **X** was used in the division operation, and then the value was decremented *after* division operation; resulting in 10/2 as the value for **Y**. This is referred to as a post-increment or post-decrement operation since the value is modified after it is used in the expression.

Conditional Expressions

You can create an expression that has a conditional element; that is, one or more elements of the expression are dependent on another value (typically a variable or Boolean expression). For example,

```
MSG = STRING(COUNT) || " STONE" || IF COUNT <> 1 THEN "S" ELSE ""
```

In this example, the string expression will add an "S" to the end of the word "STONE" if the value of COUNT is not equal to 1. If the value of COUNT is equal to 1, then an empty string is added. The **IF** clause is evaluated and either the **THEN** or **ELSE** sub-expression is used in the enclosing expression. There must always be an **ELSE** clause so there are always two possible values depending on whether the expression is true or false.

A conditional expression can be used anywhere an expression is permitted.

Arrays

An array is a list of values. The values can be of any type - they can be numbers, strings, or even other arrays. In some cases the array can be referenced just by name (such as when assigning it to another variable). You can also reference the individual elements of the array using an array subscript notation:

```
X = MY_DATA[3]
```

This references the third item in the `MY_DATA` list of values; the value in that array element is assigned to the variable `X`. You can store items in an array the same way:

```
MY_NAMES[5] = "Tony"
```

If the variable `MY_NAMES` is not already an array, the previous value is discarded and the variable re-typed as an array. If you store an array element at a position that does not exist, the array automatically extends by adding array elements containing an integer zero.

You can find out how many elements are in the array by using the **`LENGTH()`** function, as in:

```
N = LENGTH( MY_NAMES )
```

Note that the `LENGTH` function is operating on the array variable, not one of its elements. If you had used the expression **`LENGTH`**`(MY_NAMES[5])`, then you would instead get the length of the fifth element in the array.

If you print an array element, it prints just as any other scalar value. You can also print the entire array, as in the following example.

```
X[1] = "Tom"
X[2] = "Sue"
PRINT X
```

The output is `[ "Tom", "Sue" ]`, which shows an array (defined by the brackets) with two items, each of which is a string. You can create array constants like this yourself, as in the following example.

```
PERSON = [ "Tom", 35 ]
```

This creates an array called `PERSON` with two member elements. The first element is `PERSON[1]` and is a string `"Tom"`. The second element is `PERSON[2]` and is the number `35`.

You can use arrays in the addition "+" operation if the first item is an array. The second item is added to the array by making a new entry in the array. If the second item is also an array, all the elements of the second array are added to the end of the first array. For example,

```
 X = [ 3, 5 ]
 Y = [ "Tom", 35 ]
 Z = X + Y
 Q = X + 22
```

In this case, `Z` will be `[3, 5, "Tom", 35]`. The value of `Q` will be `[3, 5, 22]`.

Similarly, subtraction is used to locate and remove elements from an array (similar to using the subtraction operator on strings). The value to be subtracted can be a scalar item or another array. In either case, all elements in the value subtracted will be removed from the array being subtracted from.

```
X = [ 33, “Tom”, 5, “a”, 5 ]
Y = X - [ “Tom”, 5 ]
PRINT Y
```

The result of the array subtraction is that the array `Y` will contain `[ 33, “a”]`. Note that the subtraction removes *all* instances of the values subtracted. The original array contained two instances of the value 5, both of which were removed because 5 was in the subtrahend.

```
X = [ 101, 33.5, “Bob”, true, 32 ]
Y = X[2..4]
```

A subset of an array can be referenced using the ellipsis “..” operator. In this example, the array `Y` is set to the elements of `X` from 2 to 4, inclusive. So the array `Y` contains `[33.5, “Bob”, true]`. The range elements can be expressions used to calculate the beginning and ending elements to include in the resulting array subset. The ellipsis operator cannot be used on the left side of an assignment operation.

You can also specify the elements of an array using a second array of integer values containing selectors:

```
X = [ “Tom”, “Mary”, “Tony”, “Sue”, “Dave”, “Pam” ]
Y = [ 1, 2, 4, 5 ]
Z = X[Y]
```

In this instance, the resulting array `Z` will contains `[ “Tom”, “Mary”, “Sue”, “Dave” ]` because those are the elements of X that are identified by the selector values in the array `Y`. Similarly, the subscript for the array can be an array of boolean values which act as indicators to say if the row is to be selected or not.

```
X = [ “Tom”, “Mary”, “Tony”, “Sue”, “Dave”, “Pam” ]
Y = [ true, false, true, true, false, true ]
Z = X[Y]
```

The resulting array is `[“Tom”, “Tony”, “Sue”, “Pam”]` because those corresponding elements where true in the indicator array. This can be particularly helpful when there are a number of conditions that must be evaluated to determine if an array element (in particular an array of records) is to be selected; the indicator can be set by he analytical or selection process and then a single reference to the indicator array will dereference the selected rows.

You can use the special operator **`IN`** to detect if a value is found in an array. For example,

```
X  = [ “Tom”, “Mary”, “Tony”, “Sue” ]
      ...
IF “Mary” IN X THEN GOTO FOUND
```

In this example, the **`IN`** operator is used to determine if the value `“Mary”` is found in the array `X`, and if it is, then control is transferred to the label `FOUND`. The same basic functionality is possible by using the **`LOCATE()`** function and testing for a non-zero result.

Records

A record is another kind of list of values. The values can be of any type - they can be numbers, strings, arrays, or other records. Unlike an array where the elements are always addressed by their position in the array, a record identifies its members by the *member name*. A record can have as many members as you wish, but each must have a unique member name. Member names are identifiers, and therefore follow the same syntax rules as variable names.

In some cases the record can be referenced just by name (such as when assigning it to another variable). You can also reference the individual elements of the record using the member name of the record.

```
X = MY_DATA.AGE
```

This references the element of the record whose member name is `AGE`. The value in that record element is assigned to the variable `X`. You can store items in a record the same way:

```
MY_DATA.NAME = "Tony"
```

If the variable `MY_DATA` is not already a record, the previous value is discarded and the variable re-typed as a record. If you store a record element in a member that does not exist, the member is automatically added.

You can find out what the names of the members of a record are by using the **`MEMBERS()`** function:

```
N = MEMBERS( MY_NAMES )
```

This returns an array of strings. Each element in the array corresponds to a member name in the record. You can use one of these member names to get a value from a member indirectly using the **`MEMBER()`** function. For example,

```
      N = MEMBERS( MY_NAMES )
      X = MEMBER(MY_DATA, N[1])
```

If the first member of the record is named "`AGE`", then the above example is the same as accessing `MY_DATA.AGE`, but allows you to determine the member name as part of your program logic.

```
      X = MY_DATA[“AGE”]
```

Similarly, you can use a modified form of array syntax to access a member with a value that has been calculated by program logic. In this mode, the array subscript value *must* be a string, and that string is assumed to be a member name.

If you print a record member, it prints just as any other scalar value. You can also print the entire record, as in the following example.

```
     X.NAME = "Sue"
     X.AGE = 3
     PRINT X
```

The output is `{ AGE: 3, NAME: "Sue" }`, which shows a record (defined by the curly braces) with two items. The members are reported in alphabetical order regardless of the order in which they were created. You can define record constants like this yourself, as in:

```
    PERSON = { NAME: "Tom", AGE: 35 }
```

This creates an array called `PERSON` with two members. The first member is `PERSON.NAME` and is a string "Tom". The second member is `PERSON.AGE` and is the number 35.

Records cannot be used in arithmetic operations. However, the members of a record can be used as terms in an expression.

```
    X = PERSON.AGE + 5
```

While not being able to participate in arithmetic operations, a record can have items added to it or deleted from it using addition and subtraction operators as in the following examples:

```
    X = { NAME: “Tom”, AGE: 48 }
    Y = X + { MALE: TRUE }
    Z = Y - “NAME”
```

In this example, the record `Y` will contain `{ NAME: “Tom”, AGE: 48, MALE: TRUE }` because the “+” operator concatenates the records together. If the record being added contains fields with the same name as the record being added to, the new field values replaced the old ones. Additionally, the record `Z` contains `{ AGE: 48, MALE: TRUE }` because the `NAME` field has been deleted from the record. This requires that the value being subtracted be a string, which is a field name. It is a runtime error to attempt to delete a field that does not exist.

Record members can be arrays or records themselves. Additionally, you can create arrays of records.

```
FAMILY = [ { NAME: "Tom", AGE: 35 }, { NAME: "Sue", AGE: 3 }]
X = FAMILY[1]
Y = X.NAME
```

In this case, `Y` will have the value "Tom", since that is the `NAME` member from the first record in the array called `FAMILY`. You can create compound references of array and record names, as in:

```
FAMILY = [ { NAME: "Tom", AGE: 35 }, { NAME: "Sue", AGE: 3 }]
Y = FAMILY[1].NAME
```

This is functionally equivalent to the example above, but the array dereference of element 1 is combined with the member reference `NAME` to identify the value “Tom”.

You can explicitly specify the type of a member of a record in a constant expression by placing the type name before the field name, as in:

```
 LET X = { DOUBLE Y: 55, BOOLEAN Z: 1 }
```

This results in a double value for `X.Y` even though the constant value is an integer.

Because records can be created dynamically by subroutines, etc. the calling code may not always know which members are present. The **`MEMBERS()`** function can be used to determine the names of each member by returning an array with string names that contain the members. The **`IN()`** operator can also be used for this purpose, as show in the following example:

```
X = URL("http://apple.com/store?id=1035&item=iPod")
IF "ITEM" IN (X.QUERY) THEN CALL CHECK_ITEM(X.QUERY.ITEM)
```

In this example, the **`URL()`** function is used to parse an URL string. Since the contents of the string can be based on arbitrary input, the record identified by `X.QUERY` in the above example will contain members for each query item in the string. The second line of code tests to see if there is a record member called `ITEM` in the record, and if so uses that as a parameter to a subroutine call. Note that calling the function directly with `X.QUERY.ITEM` would result in a runtime error if the `ITEM` member did not exist, so the above statement prevents the error by conditionally calling only when the member is present.

Tables

A TABLE is a special kind of two-dimensional array in JBasic used to store data in a fashion similar to a database table. The TABLE is functionally most like an array of records where each record must have the same members and data types.

You create a TABLE using the **`TABLE`** statement, which creates a new data value in the current symbol table. The **`TABLE`** statement defines the name of the table and one or more members with a specific data type.

```
TABLE EDATA AS INTEGER ID, STRING FIRST, STRING LAST
```

This creates a table of employee data containing an employee ID number and a first and last employee name. You can add records to the table in two ways: as records or as arrays.

```
EDATA = EDATA + [101, "Bob", "Smith"]
EDATA = EDATA + [LAST: "Jones", FIRST: "Dave", ID:102]
```

In the first example, an array is being added. In this case, the array members must be in the same order as was declared in the **`TABLE`** statement. In the second case, a record is being added and the member names must exactly match the fields declared in the **`TABLE`** statement. In both cases, the data will be converted to the data type declared in the **`TABLE`** statement.

You can access a TABLE row as an array subscript. For example,

```
EMP = EDATA[1]
```

This sets `EMP` to `{ FIRST: "BOB", ID:101, LAST: "Smith" }` which is a record describing the individual row. You can of course reference a specific member using member notation:

```
    LAST_NAME = EDATA[2].LAST
```

which sets `LAST_NAME` to the string "`Jones`". Because a TABLE is meant to be used similarly to database table, you can also join two tables together with a common member value. Assume the above table `EMPLOYEES` already exists, and we create a new one with payroll data:

```
    TABLE PAY AS INTEGER ID, DOUBLE RATE
    PAY = PAY + [102, 10.50]
    PAY = PAY + [101,  7.25]
```

This creates a table with just an employee ID and a floating point value used to describe a pay rate. You might wish to join these items together to create a view of the data that matches pay rates to employees. You can use the **`JOIN()`** function:

```
    D = JOIN( EDATA, PAY, "ID")
```

This creates a new table `D` which contains the fields `ID`, `LAST`, `FIRST`, and `RATE`. The rows from the `PAY` are matched to the rows from `EDATA` using the field ID; when a row in one table has a matching row in the second table where the ID field is the same, a new row is created in the output table merging all the fields of both tables for that row.

If there is no match, then no new row is created, so if there was an employee ID 103 in the `EDATA` data but not in the `PAY` data, then no row for that employee would exist in the new table `D`.

You can sort the data in a table using the **`SORT`** statement with a **`BY`** clause that describes the column to sort by. You can also use the **`SORT()`** function on a table,

```
    D = SORT(D, "LAST")
```

This sorts the data based on the `LAST` name field. You can also subset the data with a **`WHERE()`** clause in an expression, which can follow any table expression:

```
    H = D WHERE( RATE >= 10 )
```

This creates a new table that contains the rows of the table `D` where the `RATE` field is greater than or equal to `10.0`. Rows that do not meet this criteria will not be included in the `HIGH_PAID` table. You can combine these expressions together to create complex data operations such as

```
    H = SORT( JOIN( EDATA, PAY, "ID" ) WHERE( RATE >=10 ), "LAST" )
```

This creates a table merging the pay and employee data, selecting only those with a pay rate greater than `10.00` and then sorting the result by the last name field. You can select a subset of the columns of a table using the **`SELECT()`** function, which accepts a table name and a list of column names:

```
    F = SELECT(H, "ID", "LAST")
```

An error is signaled if a field is named that does not exist. The result of the **`SELECT()`** function is a new table with just the named columns. You can use the **`WHERE`** operator as well, such as:

```
    F = SELECT( H WHERE(RATE>=10), "ID", "LAST")
```

This selects the ID and last name columns from the subset of rows where the `RATE` is greater-than or equal-to `10`. The **`WHERE`** operator can be put on the source table as shown above or applied to the result of the **`SELECT`**, but it is more efficient to use **`WHERE`** on the inner-most table of an operation to reduce the number of rows that are copies/duplicated as part of the query expression.

When you **`PRINT`** an entire table, the data is formatted for readability with column headings, etc. If you are writing an entire table to a disk file, you should use **`XML()`** format so the column definitions and rows are properly encoded and can be reconstructed into a TABLE using the **`XMLPARSE()`** function when the data is read from a file.

You can also use the **`INPUT ROW OF`** statement (an extension to the **`INPUT`** statement) to read a row of data from a file or the console.

SQL Statements

The JBasic language supports a subset of the SQL query language as part of JBasic programs. The SQL statements can manipulate the contents of TABLE data types just described, and has limited ability to include data from JDBC Databases described later in this section of the manual.

The SQL processor generally assumes that the data being manipulated (created, written, or read) resides in memory in the Table datatype. You can create a table using SQL syntax, such as

```
  CREATE TABLE FAMILY( AGE INTEGER, NAME CHAR )
```

This creates a new table named `FAMILY` that has two columns named `AGE` and `NAME`. The table has no rows of data in it yet. There are additional versions of the **`CREATE TABLE`** statement that will create a table using the contents of another table for input, or create a new table whose column names and types exactly match an existing table without copying any data from it.

Data can be inserted into a table by storing individual values or by copying data from another table.

```
INSERT INTO FAMILY VALUES( 51, "Tom" )
INSERT INTO FAMILY VALUES( 18, "Chelsea" )
```

After these statements execute, the table has two rows. The values in the **`VALUES()`** clause must be present in the same order that they appear in the Table definition.

```
INSERT INTO FAMILY VALUE({ NAME:"Sarah", AGE:16 })
```

This variation on the statement inserts a single **`VALUE`** which must represent a row of data. If the row is an array, the members of the array are processed in the same order as the column definitions in the Table. If the row is a record, there must be a record member for each possible column name in the table. After this statement executes, the table has three rows.

```
SELECT NAME FROM FAMILY WHERE AGE < 21 ORDER BY AGE
```

This statement queries the table for all rows that meet the given filter (age being less than the value `21`) and returns the data sorted by the `AGE` value. In the resulting data, only the column `NAME` is returned. The result of a **`SELECT`** statement is always a table, and if a **`SELECT`** statement is given by itself as above, the result is printed on the console. However, you can create a new table from the **`SELECT`** statement using the **`CREATE TABLE`** variation.

```
CREATE TABLE KIDS AS SELECT NAME FROM FAMILY WHERE AGE < 21
```

In this case, a new table called `KIDS` is created that only has a single column (`NAME`) and contains two rows from the source data that match the **`WHERE`** clause. In addition to SQL statements that create a new table, you can use the results of a table in an expression with the **`SQL()`** function which processes a **`SELECT`** statement and returns the resulting table. The following example finds the rows that match the age criteria and the resulting table is used to locate all members of the column for `NAME` and return them as an array.

```
KIDS = SQL("SELECT * FROM FAMILY WHERE AGE < 21")["NAME"]
```

The **`SQL()`** function returns a Table containing the rows for members whose age is less than `21`, and the resulting table is processed by the array operator for a column name. The result is an array containing all the names, which would be `[“Chelsea”, “Sarah”]`.

Global Variables

Variables are stored in a special list called the *symbol table*, which is just a list of the variables that are known to JBasic and what value they hold. Each running program has a symbol table, which contains the symbols created while that program was running. When the program stops running, the symbol table is discarded and those variables no longer have any value.

Some variables reside in a “global” symbol table, which is a table that is created when the JBasic session is started, and maintain their value regardless of whether a program is running or not. As such, values in this table are available to any program. These variables continue to have value even after a program stops running, and can be examined by a running program or by statements you type in to the JBasic command line.

Many of the variables in the global symbol table are created automatically as part of JBasic initialization. They can be used to check the state of the environment that a JBasic program is running in, or can be used to control certain aspects of how JBasic works. You use the command **`SHOW GLOBAL SYMBOLS`** to see the name and current value of all the global symbols.

Here are some useful global variables and what they are contain:

Name	Meaning
`SYS$ARGS`	The list of command line arguments that were given to JBasic when it was first started up.
`SYS$HOME`	The user’s home directory on the computer. This is a location that can be used to read or write files.
`SYS$INPUT_PROMPT`	The prompt string used by the **`INPUT`** statement if no prompt string is given. Defaults to “?”.
`SYS$LANGUAGE`	The language that the current user wishes to see messages, etc. formatted in. “EN” means English.
`SYS$STATUS`	The error code of the last error message printed.
`SYS$PROGRAMS`	An array that has the name of each *program* object loaded in memory currently.
`SYS$PROMPT`	The command prompt. Defaults to “BASIC>”.
`SYS$USER`	The username of the current user of JBasic.

About $MODE

JBasic allows a single program module to be used for multiple purposes; as a program that is executed as a **RUN** command, as a program that is called as a function, as an object-oriented method call, etc. The local variable `$MODE` is always created on behalf of the running program code to reflect how the program was invoked. Additionally, the variable `$THIS` contains the name of the current program, and `$PARENT` tells what program was active when the current program was invoked or called, and the array `$ARGS` will always contain the argument list to the current program as passed in by the caller. The possible values for the `$MODE` variable are outlined in this table:

Value	Description
`"CALL"`	The current program was invoked with the **CALL** statement. There may be arguments, and a return value may be given.
`"FUNCTION"`	The current program was invoked as a function in an expression. There may be arguments, and a return value *must* be given.
`"METHOD"`	The current program was invoked as a method for an object. There may be arguments, and a return value may be given. The local variable `THIS` identifies the object being manipulated.
`"RUN"`	The current program was invoked with a **RUN** command. There will be no arguments other than those with a default value, and no return value is permitted.
`"VERB"`	The current program is a user-written verb, executed because the verb was given as a command.

Data Files

JBasic, like most dialects of the BASIC language, allows user programs to manipulate files. In the most common case, files refer to physical data files on your computer's disk storage. They can also refer to other representations of data that *behave* like files, such as the information in a database. Files are accessed (and created, if necessary) using the **OPEN** statement, and when the file is no longer in use, the **CLOSE** statement terminates processing of the data that is in the file, residing on the computer's disk. The **KILL** statement can be used to delete the physical file if it is no longer needed.

Data files can contain text data (represented in human-readable characters) and be manipulated with **INPUT**, **LINE INPUT**, and **PRINT** statements. Files can alternatively contain binary data (an representation of information in a format that is native to the computer) and be manipulated with **GET**, **PUT**, and **SEEK** statements.

Program statements that manipulate the contents of a file use a *file reference* to indicate which file they are operating on. The *file reference* is like a variable identifier, except that it indicates an open file rather than a storage location in memory.

In JBasic, the preferred way to specify a *file reference* is to use the keyword **FILE** followed by an identifier of your choice that will be used to refer to the file as long as it is open. For example, the following program creates a file called “names.txt” on the disk, and uses the *file reference* `NAME_DATA` to refer to the file in the running program:

```
OPEN “names.txt” FOR OUTPUT AS FILE NAME_DATA
PRINT FILE NAME_DATA, “Tom”
PRINT FILE NAME_DATA, “Deb”
CLOSE FILE NAME_DATA
```

In this example, the first statement identifies the file name on the disk (“names.txt”), the mode of the file (**`OUTPUT`**) and the *file reference* (**`FILE`** `NAME_DATA`). In place of `NAME_DATA`, you could use any valid identifier name that is meaningful to you. While the file is open, the symbol `NAME_DATA` is reserved to mean the open file. Once the **`CLOSE`** statement is executed, the symbol `NAME_DATA` no longer has meaning until it is used for another purpose (to identify a file or variable).

For compatibility with other dialects of the BASIC language, you can also make a *file reference* that is an integer number. This is the way that some of the oldest versions of BASIC referenced files. Here is the same sample code as above, but using the numeric format of a *file reference* to indicate which file is being operated upon:

```
OPEN “names.txt” FOR OUTPUT AS #3
PRINT #3, “Tom”
PRINT #3, “Deb”
CLOSE #3
```

The sharp sign (#) is used to indicate a numeric file reference. The same integer number must be used for all references to the file while it is opened. No two files can have the same numeric value at the same time. However, any valid positive integer can be used as the file number (some older versions of BASIC had a limit on the numbers that could be used, but this limitation does not exist in JBasic).

See the documentation later in this guide describing the **`OPEN`** statement for more information on the use of files, file modes, and the other statements that can manipulate a file. The **`SHOW FILES`** command will list the files that are open at any given time during the JBasic session.

Database Access

A JBasic program can access external database tables as if they were files. This is done using external program elements called “drivers” to access data stored in the external database. JBasic can use drivers that support the JDBC standards (a set of definitions and requirements about how programs can access external databases) to connect to a database, send a query or request for data, and get results back to the running JBasic program.

If you do not need to access databases or are not interested in the mechanism by which JBasic and the Java runtime environment use JDBC, you can skip this section.

To be able to use a database with JBasic, you must have access to a database managed by a database server, such as Firebird, MySQL, or Oracle. Additionally you must have a JDBC driver *jar file* that contains the code that tells JBasic how to access the driver. This *jar file* is often supplied by the database vendor or can sometimes be found a open source softwares at repository sites like SourceForge.net.

Once you have acquired the driver file, you will need to make sure that it is in your *class path*. This means that it is pointed to by an environment variable like `CLASSPATH` or was specified explicitly when you ran JBasic. The provider of the driver file will usually give you examples of how to specify the location of the driver file.

When your JBasic program accesses the database, it must declare that it will be using the JDBC driver(s) that you have acquired. You do this by creating an array variable called `SYS$DRIVERS` which contains an array of strings. Each string is the name of the JDBC class, which is located in the jar file. For example,

```
    SYS$DRIVERS = ["org.firebirdsql.jdbc.FBDriver"]
```

This defines the JayBird driver for accessing Firebird databases. The actual driver class name will be different for other database drivers.

In addition to the driver definition in `SYS$DRIVERS`, you must create a data source name (DSN) record variable that describes the database you are planning to access. This is a JBasic record variable which contains specific fields that tell how to access the database. These fields are:

Field	Description
`DRIVER`	The driver name, such as "firebirdsql"
`SERVER`	The computer where the database lives, or "localhost" if it is on your own computer.
`PATH`	The database-specific path that defines which database on the server is to be used.
`USER`	The username used to authenticate the user's access to the database. If not provided, the current user's login name is used.
`PASSWORD`	The password used to authenticate to the database.

These fields are all specified in a record variable you create, and used to open a database file using the **`OPEN`** statement as shown in the following example.

```
      SYS$DRIVERS = [“org.firebirdsql.jdbc.FBDriver”]

      TESTDB.DRIVER = “firebirdsql”
      TESTDB.SERVER = “localhost”
      TESTDB.PATH = “c:\mydatabases\test.fdb”
      TESTDB.USER = “sysdba”
      TESTDB.PASSWORD = “masterkey”

      REQUEST = “SELECT * FROM FAMILY ORDER BY FIRST”
      OPEN DATABASE TESTDB AS MYDB QUERY REQUEST

LOOP:
      IF EOF(MYDB) THEN GOTO DONE
      GET MYDB AS FAMILY
      PRINT FAMILY.NAME
      GOTO LOOP

DONE:

      CLOSE MYDB
```

This example defines the name of the driver class in the `SYS$DRIVERS` array. It then creates a record called `TESTDB` which contains the required fields to define the database to access, and the user credentials (username and password) passed to the database.

It also defines a query string - expressed in the standard query language SQL - and stores it in a string variable called `REQUEST`.

Next, the **`OPEN`** statement is used to identify the database to access, the name of the file identifier to use in subsequent statements, and the **`QUERY`** clause which passes a string or variable containing the query expression.

A programming loop is specified which uses the **`EOF()`** function to determine if there is more data to be returned from the database query. If all data from the most recent query has been read then **`EOF()`** is true and the database access is terminated using the **`CLOSE`** statement. Otherwise, the **`GET`** statement is used to access a record in the database result. This record is stored in the variable `FAMILY`, and the field `NAME` from that database is printed. Note that the fields in the record `FAMILY` are defined by whatever the database query result selects as the result, so the fact that this program prints the field `NAME` depends on the programmer knowing that there is a field `NAME` in the table `FAMILY` in the database.

The query is defined by either the **`QUERY`** clause in the **`OPEN`** statement, or by subsequent **`PRINT`** operations to the database file. In either case, the text of the query is available in the file variable in a field named `QUERY`.

Interrupting Execution

It is possible to write a program in JBasic that does not end. For example, consider the following short program:

```
         PROGRAM ENDLESS
LOOP:    GOTO LOOP
```

The program named `ENDLESS` will run forever, because it has been directed to transfer control to the **GOTO** statement over and over again.

You can stop an endless loop (or any long-running JBasic program) by using the console interrupt function. Normally this is control-C on DOS, Mac OS X, and VMS consoles. On Unix systems, this usually defined by the `stty -INTR` key definition. When you press this key during a running program, a signal of **INTERRUPT** is generated for the next statement to execute. This signal causes the program to terminate and return to the command prompt with an error message:

```
BASIC> run
^C
In program ENDLESS; interrupted by user
BASIC>
```

Because the interrupt is a signal like any other, you can use an **ON** statement (documented later in this guide) to catch the interrupt request and execute a **GOTO** statement to process the interrupt in a fashion appropriate to your program, such as saving any data being processed currently, etc.

Please note that the interrupt function will not work when JBasic is being run within Eclipse, because Eclipse does not have the ability to send the native interrupt signal. In this case, just stop the JBasic thread using the "Terminate" button. Additionally, if the -nointerrupt flag was specified on the command line invoking JBasic, the interrupt function is disabled.

Debugging

A very simple program can often run correctly the first time, or with little effort to identify where an error in logic or simple typographic errors led to an unwanted outcome. But more complex programs are hard to evaluate just by inspection, and often benefit from being able to “watch them run”. This is where a debugger comes in handy. A debugger is a mechanism that allows the user to have a measure of control over the execution of a program, inspect the state of the program and variables as it runs, and detect unexpected execution paths or variable values.

JBasic includes a debugging facility. This facility allows you to trace execution of a program on a statement-by-statement basis, inspect variables, and set breakpoints, which are conditions in which the program temporarily suspends execution so you can examine information about its state.

A program is “debugged” by running it with the **DEBUG** keyword in the **RUN** command, as in the following example:

```
BASIC> run debug pi_test
Step to PI_TEST 140,
              LET OLDPI = 0
DBG>
```

This runs the program `PI_TEST` under control of the debugger. The debugger accepts command like the regular console mode, but with a prompt string of "DBG>".

The debugger stops just before executing the first non-comment statement in the program, at line 140 in the above example. That is, at the prompt above, the `LET` statement has not been executed yet.

You can use a **`LIST`** command to see the program, or **`PRINT`** a variable to see its value. You can even use a **`LET`** command to assign a new value to a variable before the next statement is executed.

You use the **`STEP`** command to execute a single statement. The **`STEP`** command causes the statement at line 140 to be executed, and then a new statement is ready for execution:

```
DBG> step
Step to PI_TEST 150,
              FOR I = 1 TO 150
DBG>
```

At this point, the value of `OLDPI` as been set to zero (the operation `of` the statement at line 140) and the program is ready to execute the beginning of the **`FOR`** loop that starts on line 150. The **`STEP`** command optionally accepts a number which indicates the number of statements to step before returning to the debugger. As a shortcut, you can just press return at the debugger command prompt to step a single line. The **`STEP`** command can also be used to control stepping *into* a function call or program call, or continuing until the called program returns. You can use the **`RESUME`** command to resume normal execution of the program. When you issue a **`RESUME`** command, execution continues until the program ends normally, or a breakpoint is encountered.

The **`BREAK`** command is used to set a breakpoint, which is a condition under which the program suspends execution and allows debugger commands to be issued. Breakpoints can be based on a program location ("stop at line 210") or be based on a condition ("stop when variable I is greater than 10"). For example,

```
DBG> BREAK WHEN I = 15
DBG> RESUME
```

The above command, when executed in the `PI_TEST` program, would resume execution of the program until the loop index variable I was equal to 15. At that point, the program would suspend execution and allow more debugger commands to be issued. See the documentation on the **`BREAK`** command for more information on how to create breakpoints, list the active breakpoints, and remove breakpoints once you no longer need them.

The **SHOW CALLS** command can be used to display the current "call stack." That is, when one program calls a second program, both are active but only one is running at any given time. So when **PROGRAM** ABC calls **PROGRAM** BAR, which in turn invokes **FUNCTION** FOO, then there is a call stack representing who has called whom.

The information presented includes the method that each "stack frame" was invoked, by displaying the $MODE variable for each frame. It also shows the name of the program at each frame, and the program statement that is currently being executed (which, for frames other than the first, will show the statement or expression that caused the next-lowest-numbered frame to be invoked). For example,

```
DBG> show calls
 1:  FUNCTION FOO        210 LET OLDPI = 0
 2:  CALL     BAR        140 LET XPI = FOO( )
 3:  RUN      ABC        190 CALL BAR( I ) RETURNS X

3 stack frames

DBG>
```

This shows that program FOO is executing at line 210. It was called from program BAR by a statement on line 140. And that program in turn was called by program ABC from line 190.

Threads

It is possible to write a program that executes more than one subroutine at a time. That is, imagine executing a **CALL** statement to run a subroutine called TASK1 and second **CALL** that runs TASK2. Instead of waiting for TASK1 to complete, the program continued immediately and invoked TASK2 as well. Each of these subroutines execute essentially at the same time.

This can be valuable in a couple of cases. If your computer has two or more processors in it (sometimes called Symmetric Multiprocessor or Multi-Core computers), then both programs actually run at the same time, and complex calculations that can be divided among more than one subroutine can complete faster by running simultaneously. Additionally, sometimes one routine must do work that waits for external events like user input, but the second routine can usefully do work anyway while the first routine waits.

In each of these cases, JBasic can be told to invoke a subroutine on a *thread*. A thread is a term indicating a context for executing code. In practice, each thread is like its own instance of JBasic running independently. Mechanisms are provided to allow the threads to communicate with each other and determine their status.

Programming with threads should not be undertaken lightly, there can be complex and occasionally surprising relationships between two or more subroutines running independently, especially since there is no direct way to control which of the threads is running at any given time, and how fast each thread completes it work.

To create threads, we add the **AS THREAD** clause on a **CALL** statement to invoke a program.

```
CALL TASK1( EMPID, 100 ) AS THREAD
CALL TASK2( EMPID, 105 ) AS THREAD

SHOW THREADS
```

This sample starts the program `TASK1` as a thread, passing it parameters. It is important to note that the parameters are processed in the context of the program that makes the **`CALL`**. So it doesn't matter if `EMPID` has a value in another thread, the value in the currently running program is the one that is used to pass to the new thread(s) as they are created. The **`SHOW THREADS`** statement will list the active and completed threads, and show their final `SYS$STATUS` value if they have completed execution.

In order for threads to communicate with each other, they can access a special file type called a **`QUEUE`**. This is created the first time it is accessed by any thread (including the main thread that started JBasic in the first place), and is deleted when all threads have closed the file. The queue acts like a message passing mechanism. Any thread can **`PRINT`** a line of text to the file and it is added to the queue. Additionally, any thread can **`LINE INPUT`** from the file and receive the oldest item in the queue's list. And the **`EOF()`** function can be used to see if there is currently anything in the queue. If a thread executes a **`LINE INPUT`** statement for a queue and there is nothing in the queue, then that thread waits until another thread puts data into the queue. If multiple threads access the same queue and perform **`LINE INPUT`** operations, there is no way to guarantee which thread will get the next item in the queue. Queues are stored in memory; when the JBasic console program terminates, all remaining data in queues is lost.

```
OPEN QUEUE “MSGS” AS F
CALL TASK1( “MSGS” ) AS THREAD(T1)
PRINT FILE F, “Here is some text”
CLOSE F


PROGRAM TASK1( QNAME )
OPEN QUEUE QNAME AS Q
LINE INPUT Q, TXT
PRINT “The message is: “; TXT
CLOSE Q
RETURN
```

The above example shows a main program and a second program used as a thread. The main program creates a queue by being the first to reference the queue. Queues are identified by a name, which is not case-sensitive. It starts the thread and passes the name of the queue that will be used to communicate with this thread. This example also shows that you can optionally specify a variable (`T1` in this case) that receives the name of the newly-created thread.

The thread starts running almost immediately, and opens the same queue. It then performs a **`LINE INPUT`** operation on that queue. If the main program has not yet put anything in the queue, `TASK1` will wait until there is data in the queue.

Meanwhile, the main thread continues running, and executes a **`PRINT`** operation to store a text message in the queue. When the `TASK1` thread receives the message, then it will print it. Meanwhile, the main program continues on its way and closes the file. Finally, the thread will also close the queue reference, which causes the queue to be deleted. You can use the **`SHOW QUEUES`** command to see the active queues that are in existence at any one time.

It is critical to note that while the above description narrates approximately what happens between the main program and the thread it creates, the order of statements executed by the main program compared to the statements in the thread is unpredictable, so you cannot write a program that assumes that a thread is at a specific line in the program unless you use a queue to suspend its execution while it waits for input.

In addition to using queues to manage information flow between threads, you can also create locks which protect critical regions of code that must not interfere with potentially concurrent execution by other threads. See the help on the **`LOCK`** statement for an example of using locks to protect threads.

Java Objects

You can directly manipulate Java objects from within JBasic, with some limitations. This is intended to support interoperability with other elements of a larger Java software architecture, such as using JBasic as a scripting agent to control interactions between other objects.

Java objects look a lot like a **`RECORD`** to JBasic, but have a few additional special properties and some limitations. Java objects are created in JBasic in one of three ways. First, a JBasic program can create a new instance of a Java object with the **`NEW()`** function. Secondly, a program that is using JBasic as an embedded language processor can store Java objects in a JBasic global variable via a method call to JBasic. Finally, external statements or functions that are executed within JBasic can create new JBasic values from any arbitrary Java object. Of these three means, only the first is documented here as it is the only mechanism done directly with the JBasic language.

To create an instance of a new object, use the **`NEW()`** function with either a string containing the fully qualified class name, or with another previously-created Java object whose class is used to create the new object. For example,

```
    MYSTRING = NEW("java.lang.StringBuffer")
```

Note that the class name must be in mixed case, and must exactly match a class name visible to the Java class loader. This means you can't arbitrarily name classes that are not included in the Java `CLASSPATH` for JBasic.

In the above example, a Java `StringBuffer` object is created. The `StringBuffer` class as defined in the Java class structure also accepts constructors with parameters. You can pass parameters to **`NEW()`**.

```
    MYSTRING = NEW("java.lang.StringBuffer", “Heading1”)
```

In this example, the constructor parameter is a string, which becomes the initial value of the string buffer. The code above is similar to `StringBuffer myString = new StringBuffer(“Heading1”);`

Note that when parameters are used with the NEW() function, the data types of the parameters must match a valid constructor for the class of object you are creating.

The above example gives a complete Class specification for the StringBuffer class. You can define a list of packages to search automatically when a partial class path is given, using the **`SET PACKAGE`** command. For example, consider the following example code:

```
SET PACKAGE="java.lang"
MYSTRING = NEW("StringBuffer")
```

In this case, the class given in the **`NEW()`** function is a partial class designation. The list of all packages that have been defined with the **`SET PACKAGE`** command are used to search for the full class definition. Packages are added to the search list with **`SET PACKAGE`** and removed with **`SET NOPACKAGE`**. You can also add packages to the list using the Java `addPackage()` method of a JBasic session object if you are using JBasic in an embedded mode. The list of packages are always visible in the array `SYS$PACKAGES` which is a read-only system variable.

You can call methods for Java objects in two ways, via the **`CALL`** statement for methods that do not return a value, or via method function in an expression.

The first example is demonstrated with this call to the `append()` method of the `StringBuffer` class. In this case, two strings are being appended to the buffer:

```
        CALL MYSTRING->APPEND("This &")

        CALL MYSTRING->APPEND(" that")
```

Note that the actual Java method name is `append` but the name we give is not case-sensitive; JBasic converts all identifiers to uppercase automatically. Because of this, you cannot call a method that has an ambiguous name if case-sensitivity is not considered. That is, in Java a method of `isMyField()` and `isMYField()` are two different methods, but ambiguous in JBasic. Such a method call will fail and you will need to consider creating a wrapper Java object with unambiguous names.

The above method call passes a string. You can pass a string, integer, double, or boolean as arguments to a method. Additionally, you can pass any arbitrary Java object as a parameter, in which case the underlying method must support the actual class of the Java object to be valid. You cannot pass arrays, chars, or other data types in Java that do not have a JBasic equivalent. Again, if you need to pass different data types, consider creating a wrapper class for the Java object that translates the supported JBasic argument types into the richer Java argument set.

You can also call a method as a function call in an expression. In this example, the `toString()` method is called to convert the string buffer created in the previous example to a standard string.

```
X = MYSTRING->TOSTRING()
```

Again, note that the `toString()` Java method must be unambiguous when converted to uppercase. The "->" notation with a method name and an argument list signals a method call. In this example, the resulting value is a string, stored in the variable `X`, that contains the text "`This & that`", created by the previous append() method calls.

Here is a more complex example that uses the Java `TreeMap` class to define a list of keys and values and keep them in sorted order:

```
TREE = NEW("java.util.TreeMap")
CALL TREE->PUT("1003", "Tom")
CALL TREE->PUT("1004", "Tony")
CALL TREE->PUT("1010", "Sue")
CALL TREE->PUT("1008", "Mary")
PRINT TREE->GET("1004")
PRINT "The Keys are:   "; TREE->KEYSET->TOARRAY()
PRINT "The Values are: "; TREE->VALUES->TOARRAY()
```

In this example, values are put in the `TreeMap` object by giving it a key and an object (in this case, both the key and the object are strings). You can use `TREE->GET("1004")` to recover the string value `"Tony"` for example. You can also use methods that return objects as shown above, where the `keySet()` method is used to return a set of the key values, and the `toArray()` method is used to convert this to an array. The resulting array is processed as a JBasic array, and the result is `["1003", "1004", "1008", "1010"]`. Because the `TreeMap` class maintains the tree in sorted order, the keys and values are always returned in the order of the keys.

Some objects allow direct manipulation of some of their fields. In this case, you can reference those fields as if they were fields in a **`RECORD`**. For example,

```
OBJ = NEW("my.domain.MyObject")
```

This creates a Java object reference to a newly created object of the class `my.domain.MyObject`, which must be loadable via the `CLASSPATH`. Let's assume that this object has a public String field called `myName` and a `public int` field called `myAge`. If you print `OBJ`, you will see these fields in the **`RECORD`** that represents the Java object:

```
BASIC> PRINT OBJ
{ MYAGE: 49, MYNAME: "Tom" }
```

Note that - like method names - the field names are converted to uppercase, and must be unambiguous when expressed in uppercase. You cannot have a field named "`age`" and one named "`Age`" in the same class or an error occurs. You can reference the field values in an expression, as in:

```
    OBJ.MYAGE=50
    PRINT OBJ.MYAGE
```

Unlike a JBasic **RECORD**, you cannot create new fields by assigning a value to them; you can only reference fields that actually exist in the Java object. The **MEMBERS()** function will return a list of the fields in the Java object just as it would in a **RECORD**.

If the Java class has static fields, these are available in the Java object but are not normally printed out or included in the **MEMBERS()** list. To reference a static field in the Java object, you must precede its name with an "_". Suppose the `MyObject` class used in the above example includes a static int value of `adultAge` with a value of 18. You would not see it in the **PRINT** command of the object, but you could reference it as `OBJ._ADULTAGE`. If you use the optional second parameter to the **MEMBERS()** function you can see the list of static fields as well as the object fields, as in **MEMBERS(**`OBJ,TRUE`**)** which will include the static field names in the list.

When JBasic calls object methods that throw a Java exception, this is reported back to the JBasic program as a `OBJEXCEPT` error code, with the status argument being a text representation of the underlying Java exception. Also note that if a method is called as a function and does not return a value, then an error is thrown. A method that returns a value can always be executed with the **CALL** statement, if a value is returned but not used it is just discarded.

The function **METHODS()** can be used to return a string array that contains text descriptions of the methods that can be called on a given object. These descriptions are in Java notation and include the parameter types and return value types. The function **CLASSOF()** can be used to determine the class of the object, or will return the underlying Java type if the argument is a JBasic value other than an object.

JBasic Objects

JBasic implements limited support for experimenting with object-oriented programming natively in the JBasic language. JBasic objects are special cases of **RECORD** data types. JBasic allows you to create hierarchies of classes as well as storage containers. The following section describes how this can be used in JBasic programs. This section *does* not attempt to explain object oriented programming!

JBasic objects are **RECORD** values with additional information describing what *class* and what *container* they belong to. The *class* distinction is a conventional object paradigm; the class provides additional information about the fields and methods (programs) that can be invoked on behalf of the object. The *container* associates an object with another object, such that you can access fields in the current object, or in the container object (or the container's container object, etc.).

Consider the following example code, which illustrates using JBasic objects to represent employees and a manager of a department in a company:

```
CLASS EMPLOYEE (SALARY AS DOUBLE, NAME AS STRING)
CLASS MANAGER SUPERCLASS EMPLOYEE ( ACCESS AS STRING )
CLASS DEPARTMENT (DNAME AS STRING)

CUSTSERV = NEW(DEPARTMENT)
CUSTSERV.DNAME = “Customer Service”

BOB = NEW(MANAGER) OF DEPARTMENT
BOB.NAME = “Bob Smith”
BOB.SALARY = 65000
BOB.ACCESS = “BUILDING 33”

SUE = NEW(EMPLOYEE) OF BOB
SUE.NAME = “Sue Jones”
SUE.SALARY = 12.50

CALL BOB->PAY()
CALL SUE->PAY()
PRINT “Sue is in the “; SUE->DNAME; “ department”
```

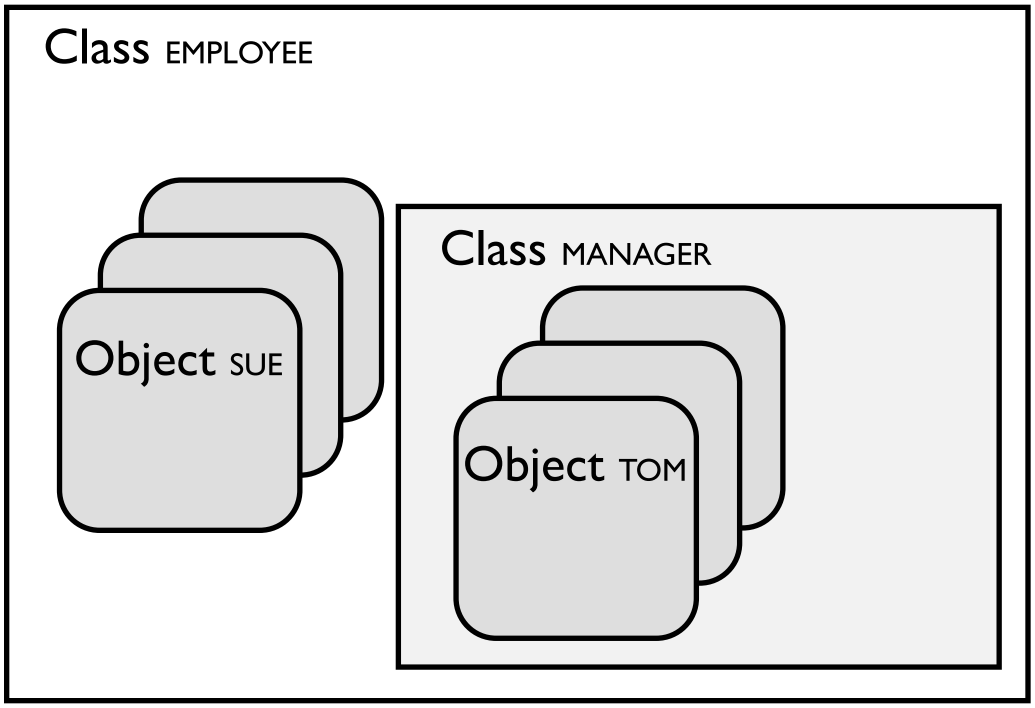
The first section defines three *classes*. A *class* is a special kind of object that is used to define the characteristics of other objects. For example, the *class* EMPLOYEE has two fields named SALARY and NAME. The class MANAGER is a class that has EMPLOYEE as its *superclass*. That is, MANAGER has all the fields of EMPLOYEE, but also the additional fields given in this class (ACCESS, in this example).

The next section creates a DEPARTMENT object. This is an instance of a DEPARTMENT, and has all the fields of any DEPARTMENT. The next statement sets the field value for this specific department.

Next, a MANAGER object is created named BOB. The **OF** clause means that this manager is contained by the DEPARTMENT object. An object can be contained by at most one other object. The fields of BOB are then filled in, including the salary which (for objects of type MANAGER) is defined as $65,000 a year. The BOB object has a field that EMPLOYEE objects do not; the ACCESS field is set to specify the building this object has access to.

The next block creates an EMPLOYEE named SUE. BOB is the container object for SUE, which means that some kinds of references for fields in SUE will search BOB and then CUSTSERV because that is the *container* hierarchy. Objects of type EMPLOYEE are paid by the hour, so her salary is expressed as $12.50 an hour.

The next section illustrates method invocation to perform work on an object. In both cases the same method is invoked (PAY) on all the objects regardless of whether they are EMPLOYEE or MANAGER objects. The underlying implementation in JBasic uses the *class* information to locate a method. For BOB, a program named MANAGER$PAY is searched for first. If found, it will be called to arrange to pay BOB (presumably by knowing that his salary is annual and not hourly). If there was no *method* called MANAGER$PAY, then JBasic would search for EMPLOYEE$PAY and call it if it was found. Finally it would search for a program OBJECT$PAY which is the root *class* of all objects. Whenever a program is located, it will be called and the current object (BOB or SUE) will be passed to the program as a local variable that is always called THIS. So the program could reference the salary using THIS.SALARY.



When `SUE` has the same `PAY` method invoked, it does not look for `MANAGER$PAY` but instead starts at the class for the `SUE` object which is `EMPLOYEE`, and locates `EMPLOYEE$PAY` which presumably pays her based on an hourly rate.

The last section in the sample code shows a container field reference. If the reference was to `SUE.DNAME` then it would look for a field in `SUE` called `DNAME` to print. However, the -> operator means that JBasic will search the container hierarchy to locate a field called `DNAME`. In this case, it searches `SUE` first, then looks in `BOB` (which is the container for `SUE`) and then looks in `CUSTSERV` which is the container for `BOB`. It finds the field `DNAME` here, and prints the name of the department that both `BOB` and `SUE` are members of.

XML Data

XML refers to eXtensible Markup Language, and is an industry standard way of expressing information that can be shared between programs, computers, and networks with great accuracy regardless of the computer systems, languages, etc. involved. JBasic can express any data value as XML, and can parse XML created to describe JBasic values.

Here is an example of the most basic structure if an XML Value definition:

```
<?xml version="1.0" encoding="UTF-8"?>
   <JBasicValue>
     <Integer>3</Integer>
   </JBasicValue>
```

This string represents the integer value 3. The first part is a comment that identifies an XML string. The second part is common to all JBasic XML strings, and indicates that this XML code is a JBasicValue, which means a data representation for JBasic. The item <JBasicValue> is an XML header tag, and is the default tag for identifying a JBasic value. You can specify a different header tag value in cases were you use the built-in functions for creating or processing XML data, discussed later in this section.

The header tag followed by the actual data type and value, and then the matching terminator for the `JBasicValue` tag. This same format applies to all the scalar data types, shown in the following table.

Data Type	Example XML Representation
BOOLEAN	`<Boolean>true</Boolean>`
DOUBLE	`<Double>3.553</Double>`
INTEGER	`<Integer>3</Integer>`
STRING	`<String>"This is \"Test\""</String>`

Note that the string value representation is in quotation marks, and contains any characters that cannot be represented in a string as escaped characters, such as the \" which represents a quotation mark.

Arrays are stored as a list of values. The next example is an array of integer values, containing `[101, 102, 103]`, expressed as XML:

```
<?xml version="1.0" encoding="UTF-8"?>
   <JBasicValue>
     <Array count="3">
       <Integer>101</Integer>
       <Integer>102</Integer>
       <Integer>103</Integer>
     </Array>
   </JBasicValue>
```

In the `<ARRAY>` tag, the attribute `count` tells how many members are in the array that is being represented here, and is followed by that many individual data representations.

A **`RECORD`** type is a little more complicated because it uses the field names as the tags around each value. Here is the XML representation of a simple record `{AGE:14, NAME:"Susan"}`

```
<?xml version="1.0" encoding="UTF-8"?>
   <JBasicValue>
     <Record>
       <AGE>
         <Integer>14</Integer>
       </AGE>
       <NAME>
         <String>"Susan"</String>
       </NAME>
     </Record>
   </JBasicValue>
```

Here, the **`<Record>`** tag is followed by each field name as a tag containing a data item.

Finally, you can combine types in XML just as you can in JBasic values to create compound data types. Below is an array of records each describing a person:

```
<?xml version="1.0" encoding="UTF-8"?>
   <JBasicValue>
     <Array>
       <Record>
         <AGE>
           <Integer>14</Integer>
         </AGE>
         <NAME>
           <String>"Susan"</String>
         </NAME>
       </Record>
       <Record>
         <AGE>
           <Integer>10</Integer>
         </AGE>
         <NAME>
           <String>"Danny"</String>
         </NAME>
       </Record>
     </Array>
   </JBasicValue>
```

You can generate valid XML for any given value by using the **`XML()`** function, which returns a string. You can parse valid XML that describes a `JBasicValue` object using the **`XMLPARSE()`** function. See the individual documentation on these functions later in this manual for more information, including how to use non-default XML tags to identify JBasic values.

Additionally, you can save a program as an XML definition by using the **`SAVE XML`** command. When you load a program file, JBasic checks to see if the file is an XML definition of a program and parses the file accordingly. See the documentation on the **`SAVE`** command for more information.

Finally, you can use the **INPUT** statement to read XML expressions from a file, either as raw XML data or by converting them to JBasic values. See the documentation on the **INPUT** statement for more information.

Macros

JBasic includes a rudimentary *macro facility* which allows source code to be dynamically modified as it is read and processed, using substitution values set up by the user. This can be used to create program templates that are incomplete programs where strings are filled in as the program is loaded to create the complete program.

```
FUNCTION <@FUNC_NAME@> ( X )
RETURN X * 2
```

This simplistic program uses a macro `FUNC_NAME` to define the name of the function. The actual name is not resolved until the program is loaded into memory. A program can be written to perform this operation dynamically.

```
LET<MACRO> FUNC_NAME = "TIMES2"
LOAD "function_template.jbasic"
```

This program sequence creates a macro variable (identified by storing the symbol in the reserved table name `MACRO`) with the name of the function. The second statement then causes the file described above to be loaded into memory. When it is loaded and compiled, the macro substitution will occur and the resulting function name will be called TIMES2, and can be called and executed after the LOAD command complete.

The macro substitution operation occurs only once when the program file is loaded (or a statement is typed in at the command prompt). The macro variable can have it's value changed for a subsequent **LOAD** of the same template file and a new function will be created with the new name. You can use the **SHOW MACRO SYMBOLS** command to see all the macro symbol definitions.

The default *macro quote* characters are `<@` and `@>`. The expression between these symbols is used to define the substitution value; it can be any expression that references a constant value or a macro symbols; other symbols such as local program variables are not recognized as macro symbols.

If your program needs to use the character strings "`<@`" or "`@>`" as literal text instead of macro symbols, you can change the macro quote characters or disable macro substitution entirely.

```
SET MACRO_QUOTES=["<<", ">>]

SET NOMACRO_QUOTES
```

The first example sets a different string to represent the starting and ending macro quote characters. Anytime these characters are found in program source, they are used to replace the quoted value with a substitution string. If the substitution operation references unknown variables or has an expression syntax error, then the substitution is just an empty string.

The second statement above disables the macro facility entirely by specifying that there are no macro quote characters to be searched for.

Finally, you can use the **`SET LOGGING=3`** command to enable diagnostic logging of JBasic operations, which includes showing when macro substitutions occur and also reporting details of any expression errors in the macro strings. This can be used when debugging a template file, but should be turned off with a **`SET LOGGING=1`** command to restore normal error reporting when you are done debugging a template file.

A Complete Sample Program

The following is a complete sample program written in JBasic. You can type this in and run it yourself, or make changes to expand its functionality. The program's job is to let the user enter a list of grades, and produce a simple average.

The program accepts the input as a list of numbers. The user tells when the list of numbers is complete by entering the word "END" as the value. The program inputs each of the values as a string, and then converts them to a number to perform the average computation.

```
100         PROGRAM AVERAGE
105   //    Sample Program AVERAGE
110   //    Version 1.0
120   //    By Tom Cole
125   //
128   //    Initialize the summation variables
130         SUM = 0.0
140         COUNT = 0
145   //
147   //    Get the next input value
150   GET_NEXT:
160         LINE INPUT “Enter grade or END: “; GRADE
162   //
165   //    Are we done?  Make sure “end” and “END” are
167   //    treated the same by uppercasing user value.
170         IF UPPERCASE(GRADE) = “END” THEN GOTO DONE
173   //
175   //    Convert the string to a number, and sum up
180         NUMGRADE = DOUBLE(GRADE)
190         ADD 1 TO COUNT
200         ADD NUMGRADE TO SUM
210         GOTO GET_NEXT
212   //
215   //    If done, calculate and print average
220   DONE:
230         AVE = SUM / COUNT
240         PRINT USING “The average is ###.##”; AVE
242   //
255   //    All done, program stops here
250         RETURN
```

The Rest Of This Document

The next sections of this manual will describe the operation and syntax of each of the statements in the JBasic language. This is followed by a description of each of the supplied runtime functions in the language, and a description of how MultiUser server mode is used.

The final part of the manual will describe tools that are used to help understand the internal operation of JBasic, and how to extend JBasic so that it has additional statements that you create, or can call code you write from within JBasic programs.

The Legal Stuff

JBasic is open source. The author, Tom Cole, retains licensing control. JBasic is meant to be used freely by anyone who finds value in it. It is licensed under the GNU General Public License 2.0. A copy of this license can be found at:

http://www.gnu.org/licenses/old-licenses/gpl-2.0.txt

In a nutshell, you are free to use JBasic in any non-commercial way that you want, as long as you keep a reference to the authorship and license of JBasic visible in your product splash screen, credits, or documentation. You are NOT free to take JBasic and incorporate it into a commercial product without direct written permission of the Author(s).

JBasic incorporates code from the open source project TelnetD, at http://sf.net/projects/telnetd. This source is authored by Dieter Wimberger and is licensed under the BSD License and the GNU Lesser General Public License (LGPL). The full text of this license can be found at:

http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

This user’s guide is Copyright 2006, 2007, 2008, 2009 by Tom Cole. All rights are reserved. You may reproduce this document for educational or personal use, but may not reproduce it for commercial purposes of any kind without written permission of the Author.

Where to go from here?

Well, the most obvious thing is to just try it out and see if you find it useful. If you find a problem, or have something you want JBasic to be able to do, please feel free to email me with questions or a description of your problem at tomcole@users.sf.net

You can always get the most recent source code for JBasic from the SourceForge server, along with occasional packaged releases:

http://jbasic.sf.net

This document is authored using Apple Computer’s *Pages*, a word processing and page layout program for Mac OS X. The PDF file is generated using the Mac’s built-in PDF export functions. Please report any errors in the document (including errors in formatting or readability using any specific platform or document reader) to tomcole@users.sf.net.

This page intentionally left blank.

ADD

The **ADD** statement adds the value of an expression to an existing variable.

```
      ADD (HOURS*RATE) TO PAY
```

The above statement calculates the expression and adds it to the existing value of `PAY`. If the variable `PAY` does not exist, then this statement generates a runtime error. This is essentially equivalent to the statement **LET** `PAY = PAY + (HOURS*RATE).`

Note that if this expression uses pre- or post-increment or decrement operators in the target expression, they are processed after the statement completes the storage of the new value in the target. For example,

```
    B = 3
    ADD X+B TO EMPS[B++]
```

The increment operation of `B` will take place after the sum `X+B` is stored in the array `EMPS` at the location of the current value of `B` (3 in this case).

The **ADD** statement can also be used to add rows to a table, as in this example:

```
    TABLE EMPS AS INTEGER ID, STRING NAME
    ADD [101, “Tom”] TO EMPS
    ADD [351, “Sue”] TO EMPS
```

Each array of values is added as a row at the end of the given table.

ARRAY

The **ARRAY** statement declares one or more variables as array variables, and optionally creates an array of arrays. For example,

```
ARRAY X[10]=TRUE
```

This statement creates an array X with ten elements, each of type BOOLEAN and each element initialized to the value given. Unlike other type declaration statements like **BOOLEAN** or **INTEGER**, this statement uses the type of the initial value to set the type of each array element.

If no initial value is given, the array is created as an *array of arrays.* That is, each array element in the named variable will itself be an array, containing no values. For example,

```
ARRAY Y[3]
Y[1,2] = 55
```

This first creates an array Y each containing the value `[]`. Each array element can then be extended by assigning additional values, so the second statement extends the first array to have two members, and assigns the integer 55 to the array. The resulting array looks like `[ [ 0, 55 ], [ ], [ ] ].`

In most cases, you would use the specific declaration statement for the type of element(s) you wish to define in an array. Use the **ARRAY** statement when you want to pre-define an array of arrays.

```
ARRAY X
```

You can omit the array index size completely and the result is the creation of an empty array, equivalent to issuing the statement **ARRAY** `X[0]`

BOOLEAN

The **BOOLEAN** statement declares one or more variables as Boolean variables, and assigns them initial values. Optionally, the value can be declared as an array of Boolean values by using an array size. For example,

```
BOOLEAN X[10], FLAG=TRUE
```

This statement creates an array with ten elements, each of type BOOLEAN. The array is initialized to the default value for the type, so in this case it is an array of false values. The array size must be specified using square brackets, not parenthesis.

The statement also creates a single BOOLEAN value named FLAG, which is initialized to a specific value of true.

This statement has the same effect of using the **DIM** statement with an explicit type of **BOOLEAN**, but also allows you to specify an initial value for each item.

BREAK

The **BREAK** command is used to control breakpoints in the debugger. See the introductory section on debugging to get more information on how to run a program under control of the debugger. The **BREAK** command can only be used when the debugger is active. The **BREAK** command has several formats, each of which are described here.

```
BREAK AT line-number
```

This specifies a breakpoint at a given line number in the program. When this line is about to be executed, the debugger will gain control of the running program and let you execute new debugger commands. The breakpoint is triggered each time the given line is executed.

```
BREAK WHEN expression
```

This causes a breakpoint when the expression evaluates to `true`. When it becomes true the first time, then a breakpoint is set and the debugger lets you enter debug commands. The breakpoint will not be hit again until the expression goes from false to true. A break point set using **BREAK WHEN** `IX=5` will halt execution at the moment when variable `IX` becomes equal to 5, but won't halt execution continually as long as variable `IX` is still 5. It will only break when variable `IX` changes from “not 5” to “5”. This is referred to as “triggering” the breakpoint.

When a program is running under control of the debugger, two variables are automatically set at the start of each statement. These are _LINE_ which is an integer containing the current line number, and _PROGRAM_ which contains the name of the current program. You can use these variables in a **WHEN** condition to create more complex breakpoints than can be specified with the **BREAK AT** format, such as

```
   BREAK WHEN _LINE_ > 150 AND IX = 3
```

This will trigger a breakpoint when the program has executed past line 150, and the value of the variable IX becomes equal to 3.

You can use the RESUME RESET command to resume execution of a program after resetting all breakpoints to the “untriggered” state, which means that the next statement will cause each breakpoint to be evaluated to see if it becomes triggered.

You can see a list of the breakpoints you have specified and their name using the **BREAK LIST** command.

```
DBG> BREAK LIST
      BREAK_1: WHEN I > 5 [true]
      BREAK_2: AT 150

DBG>
```

Each breakpoint is given a name when it is created, in the form BREAK_*n* where "*n*" is a sequence number. The condition of the break point (an **AT** location or a **WHEN** condition) is indicated. For the condition, if the flag [true] is printed next to the condition, it means that the condition has been met and a break already taken.

The **BREAK LIST** output is useful for determining the identifiers for breakpoints to be used with the **BREAK CLEAR** command.

```
  BREAK CLEAR name
  BREAK CLEAR ALL
```

This lets you clear (remove) a specific breakpoint by name - usually determined by the **BREAK LIST** command - or to remove all breakpoints.

CALL

The **CALL** statement allows one program to execute another. The **CALL** statement suspends execution of the current program, executes another program (optionally passing in arguments to it) and then resumes execution of the current program once the called program completes execution.

```
CALL name [ ( arg1 [, arg2...)] [RETURNS variable]
```

The name of the program is usually an identifier that is the name of the *program* that you are calling. It can also be written as

```
CALL USING( expression ) ...
```

In this case, the expression is evaluated, and the resulting string value is treated as the name of the program to call. This allows you to put the name of the program in a variable and use that, for example.

If there are arguments to pass to the program, they are given in parenthesis after the program name. The number of arguments must be the same or greater than the number of arguments in the **PROGRAM** declaration. See the documentation on **PROGRAM** for more information about how arguments are stored in local variables in the called program.

Finally, a called program can return a value to the calling program, using the **RETURN** statement. In this case, the return value will be stored in the **RETURNS** variable. This is the name of a variable such as you would use in an assignment (`LET`) statement, and after the program is run the value is set for use in the calling program. A program called this way can also be called as a function in an expression, as long as the program name is not also the name of a user-written or built-in function.

The return variable can specify *scope qualifiers* to define the attributes of the variable. See the documentation on the **LET** statement for more information on scope qualifiers.

When a program is invoked via a **CALL** statement, the variable `$MODE` in the called program's symbol table is set to the value `"CALL"`. This is used to differentiate between programs executed by **CALL** versus **RUN**. If the program is being called as a function, then `$MODE` is set to "`FUNCTION`".

```
   CALL PROCESS_DATA( RECORDID, “LOAD” ) AS THREAD(TID)
```

As documented in the introductory section on threads, you can add the **AS THREADS** clause which causes the called program to run in an independent instance of JBasic, possibly at the same time as the main program. In this example, the name of the new thread is returned in the variable `TID`. When run as a thread, it is not possible to get a return value directly from the thread. Use a queue to pass information between threads or between a thread and the main program. See **OPEN** for more information.

CHAIN

The **CHAIN** command transfers control from the current program to a new program, identified in the command. Optionally, a starting line number in the new program is given to indicate where the program is to begin execution.

```
    CHAIN "program" [, line-number]
```

The program to transfer control to must be a program already loaded from a workspace. The program name is a string expression, so you can construct a variable or other expression to define the name as well as using a string literal.

If you wish to specify a line number in the program at which to begin execution, put that after the program name (separated by a comma).

The current program stops execution, and the new program is run instead. Only symbols in the current program that are part of a **COMMON** definition are available to the new program. See the documentation on the **COMMON** statement for more information.

When a program is run via a **CHAIN** statement, the `$MODE` variable is set to the string "`CHAIN`" (as opposed to "`CALL`" or "`RUN`"). In addition, a variable `$CHAINED_FROM` is created that has the name of the program that executed the **CHAIN** operation. The `$PARENT` variable is the same as the parent of the program that executed the **CHAIN** operation.

Consider the following example programs:

```
 10  PROGRAM CHAIN1
100  COMMON NAME$
110  LET NAME$ = "Tom"
120  LET AGE = 47
130  CHAIN "chain2", 100
140  PRINT "Never returns from chain!"
```

```
 10  PROGRAM CHAIN2
 15  PRINT "This won't be run!"
 20  END
100  PRINT "The name is "; NAME$
130  PRINT "Done with CHAIN2 program."
```

In these examples, the program `CHAIN1` will run, and will then be replaced with the code for `CHAIN2`. This is different from a **CALL** statement, where the program that is doing the calling continues to run after the called program returns. In a **CHAIN** operation, the new program replaces the old program in terms of execution.

In the example, the **COMMON** statement is used to define a variable that is *common* to all programs that are executed via a **CHAIN** from the current program. This means that the variable `NAME$` will be available in the program `CHAIN2` and will retain its value. The variable `AGE` will not be available to program `CHAIN2`. See the documentation on **COMMON** for more information.

Note that after the **CHAIN** statement is executed in the first program, control will never return to this program - the final **PRINT** statement will never be executed. In addition, the **CHAIN** statement specifies a starting line number in the destination program, so the first **PRINT** statement in the chained-to program will not be executed.

CLEAR

The **CLEAR** command is used to remove a stored value or program element from memory. The statement has a variety of formats, depending on what you are trying to clear:

```
CLEAR PROGRAM name [, name…]
CLEAR FUNCTION name [, name…]
CLEAR VERB name [, name…]
CLEAR TEST name [, name…]
CLEAR FIELD file-identifier
CLEAR SYMBOL name [, name…]
CLEAR LOCK name [, name]
```

In all the above cases, the name of the item is given as an identifier, such as

```
CLEAR SYMBOL MYNAME
```

An error is signalled if the named object does not exist. Otherwise, it is removed from memory.

For program objects (**PROGRAM**, **FUNCTION**, **VERB**, and **TEST**), the object can subsequently be recreated by a **LOAD** command to re-load program elements. A symbol can be recreated by assigning a value to the symbol name or by using the **DIM** statement.

See the **SHOW** command for information on how to see what program objects or symbols exist at any given time in the JBasic session.

You can specify more than one object to be cleared by using a comma-separated list, such as

```
CLEAR PROGRAM FOO, BAR
```

This clears both program **FOO** and program **BAR** from program memory. All objects in the list must be of the same type, i.e. they are all **PROGRAMS** or **SYMBOLS**, etc.

In addition to the above uses of the command, you can use the **CLEAR THREADS** command to remove references to threads that have stopped. Because threads run independently of the main program, they may complete at any time. You can use the **THREADS()** function to get a list of all threads, and the **THREAD()** function to get a record of information about a specific thread. When a thread completes it is not deleted so the main or creating thread can get information about its completion status, final error code if any, etc. Because of this, threads must be explicitly cleared when they are no longer needed. The **CLEAR THREADS** command deletes all competed threads. Threads that are still active are not deleted.

CLOSE

The **`CLOSE`** command closes a file accessed via an **`OPEN`** statement. The file identifier used in the **`CLOSE`** statement must be the same one that was returned from the **`OPEN`** statement.

```
CLOSE [ FILE ] identifier
```

The keyword **`FILE`** is optional. The keyword **`FILE`** can also be expressed as a *sharp sign* ("#") for compatibility with other dialects of BASIC.

If the statement is given without a file identifier, then all files opened by the user's program(s) are closed. Files opened by JBasic itself (such as the console) are not closed by this operation.

```
CLOSE
```

For compatibility with other dialects of BASIC, you can specific **`CLOSE ALL`**, which is the same as **`CLOSE`** with no additional file identifier. See the information on the `OPEN` command for information on how files are opened and accessed. You can also use the **`SHOW FILES`** command to display the list of currently open files.

COMMON

The **COMMON** statement is used to declare variables in the current program that will also be available in any program that is executed by a **CHAIN** statement from this program. The **COMMON** statement has the same syntax as the **DIM** statement.

```
COMMON variable [AS type] [, variable [AS type]...]
```

Variables declared in this way have the *common* attribute, and are copied to the symbol table of any program that is run via a **CHAIN** statement. All other symbols in the current program are discarded during a **CHAIN** operation.

You can also mark a variable as being a **COMMON** variable by setting the `COMMON` scope qualifier attribute in an assignment statement.

```
LET MYDATA/COMMON/ = "Data to pass to another program"
```

The `/COMMON/` tag indicates that the variable is marked as being common to the chained program(s). Once a variable is marked as being common, it cannot be unmarked.

See the documentation on the **CHAIN** command for more information.

DATA

The **DATA** statement is used to store data elements directly in the program. These **DATA** elements can be loaded into a variable using the **READ** statement.

```
DATA 3, "Tom"
DATA {age:40}
```

The above statements define three constant values. One is an integer, one is a string, and one is a *record* type. When a **READ** statement is executed, it will read one or more of the data elements in the program into variables, which can then be used just as if they came from a **LET** or **INPUT** statement.

A **DATA** statement can only contain constant data, such as numbers or strings. However, you can include expressions as long as all of the parts of the expression are constants - that is, they contain no variables as part of the value expression.

DATA elements can be stored anywhere in the program. When a **READ** statement is executed, it reads starts with the first **DATA** statement in the program and reads the first item in that **DATA** statement. The next **READ** uses the next **DATA** element, either in the same **DATA** statement or the next one in the program. When all the **DATA** has been read, the next **READ** statement will read the first item again.

A **DATA** statement cannot be executed, and cannot be used except in a program (you cannot use the **DATA** statement on the command line, for example). Because a **DATA** statement is not executed, the physical order of the **DATA** statements in the program is all that matters in determining which **DATA** element(s) to read next - you cannot use a **GOTO** or other statement to change the order in which **DATA** is accessed by the **READ** statements.

Use the **REWIND** statement to reset the location where the next **READ** statement gathers its data. The **EOD()** function can also be used to test to see if there is more **DATA** to be read.

See the documentation on **REWIND**, **READ**, and **EOD**() entries for more information.

DEFFN

The **`DEFFN`** statement is used to define a local statement function. This is a function that is expressed as a single expression, and is available for use only in the program that defines it.

```
100   DEFFN LINE( X, B ) = 3 * X + B
              ...
230   Y = LINE( XPOS, 10 )
```

In this example, a function `LINE()` is defined which takes two parameters. The parameters are used in the expression that defines the behavior of the function. This expression can be any valid JBasic expression, including calling other functions.

The second statement shows calling the function. The values `XPOS` and `10` are substituted for the parameters `X` and `B`, and the expression is evaluated. The result is used as the function result, and stored in the variable `Y` in this example.

Statement functions can only be defined using expressions, so no flow-of-control is possible in a statement function. They are intended purely to compute straightforward mathematical expressions.

A statement function is only available in the program that defines it. That is, if the above code is located in program `FOO`, then a call to the function `LINE()` from program `BAR` will result in an unrecognized function call error. Only program `FOO` can call the functions that it defines.

DELETE

The **DELETE** command deletes lines from the current program. You can list a specific line, or a range of lines. If the beginning or ending of the range is not given, then the first or last lines are assumed.

```
    DELETE 1150                  // Delete line 1150
    DELETE 100-150               // Delete lines 100..150
    DELETE -30                   // Delete lines up to line 30
    DELETE 500-                  // Delete lines after 500
```

You can use statement labels for line numbers as well. The label must exist or an error is reported.

```
   DELETE INIT - TERM            // Delete lines between INIT and
                                 //    TERM statement labels.
```

If you specific **DELETE** without giving a range of lines, the entire program will be deleted from the stored program memory. If you delete the entire program, there is no "current" program until the next **OLD** or **NEW** statement, or until you start to enter new program lines.

DIM

The **DIM** statement declares a variable and gives it a type. This is usually used to create an array (the word **DIM** is a contraction for the word DIMENSIONs), but can be used to create any variable without having to assign a specific value to it.

```
DIM INTEGER X[10]
```

This statement creates an array with ten elements, each of type **INTEGER**. The array is initialized to the default value for the type, so in this case it is an array of zeroes. The type value must be one of **BOOLEAN**, **STRING**, **DOUBLE**, or **INTEGER**. The array size can be specified in brackets or parenthesis, for compatibility with other dialects of BASIC.

If you omit the size value in parenthesis, then a simple variable is created of the given type.

```
DIM BOOLEAN FLAG
```

This creates a variable `FLAG` of type Boolean, and sets it to false by default. This has the same effect as using the **BOOLEAN** statement.

You can specify more than one variable in a single **DIM** statement by using a comma:

```
DIM INTEGER A, STRING B, BOOLEAN C[10]
```

This creates three variables: an integer `A`, a string `B`, and an array of Boolean values called `C`.

You can use the **DIM** statement to create multi-dimensional arrays. For example, this creates an array named `BOARD` that is a 3x3 array of integers:

```
DIM INTEGER BOARD[3,3]
```

The result is identical to setting `BOARD` to the value `[[0,0,0],[0,0,0],[0,0,0]].` Each element of the two-dimensional array is initially filled with the integer value zero.

DIVIDE

The **DIVIDE** statement divides an already-existing target value by the value of an expression.

```
    DIVIDE HOURS BY 60*60   // Convert hours to seconds
```

The above statement calculates the expression and divides the existing value of HOURS by that value.. If the variable HOURS does not exist, then this statement generates a runtime error. This is essentially equivalent to the statement **LET** HOURS = HOURS / (60*60).

Note that if this expression uses pre- or post-increment or decrement operators in the target expression, they are processed after the statement completes the storage of the new value in the target. For example,

```
   B = 3
   DIVIDE EMPS[B++] BY X+B
```

The increment operation of B will take place after the division by X+B is stored in the array EMPS at the location of the current value of B (3 in this case).

DO

The **DO** statement creates a loop that runs until an expression results in either a true/non-zero value or a false/zero value, depending on the form of the loop.

```
DO

   ...statements...

UNTIL Boolean-expression
```

The *Boolean-expression* is any expression that results in a Boolean (true/false) or numeric value. If the result is false or zero then the loop runs again. If the result is true or non-zero, then the loop exits and execution of the program continues after the **UNTIL** statement.

```
DO

   ...statements...

WHILE Boolean-expression
```

The *Boolean-expression* is any expression that results in a Boolean (true/false) or numeric value. If the result is true or non-zero then the loop runs again. If the result is false or zero, then the loop exits and execution of the program continues after the **WHILE** statement.

Loop statements such as **DO...WHILE** or **DO...UNTIL** cannot be executed directly from the command line, but are only valid in running programs.

You can place the condition at the "top" of the loop so it is evaluated before the loop runs the first time, as in the following example:

```
X = 10
DO WHILE X > 0
   PRINT X
LOOP
```

This is similar to the **DO..WHILE** loop above, except that if the value if `X` is less than or equal to zero instead of having the value 10, the loop will not execute at all. You can express **DO UNTIL..LOOP** the same way if the logic of your loop calls for evaluating the expression at the "top" of the loop instead of the "bottom."

You can use the **CONTINUE LOOP** and **END LOOP** statements to change flow of control while executing the body of the loop. Consider the following example:

```
I = 0
DO WHILE I < 10
    I = I + 1
    IF I = 5 THEN CONTINUE LOOP
    PRINT "Value "; I
LOOP
```

In this case, when the value of the index variable is 5, the **CONTINUE LOOP** statement causes control to transfer back to the top of the loop body, which retests the **WHILE** condition - it discontinues the current instance of the loop body, and runs the next loop iteration. If the ending condition is met, then the loop exits normally. In the above example, all the index values from 1 to 10 will be printed except the value of 5, because the **CONTINUE LOOP** causes the rest of the loop body to be ignored.

```
I = 0
DO
    I = I + 1
    IF I = 5 THEN END LOOP
    PRINT "Value "; I
UNTIL I >= 10
```

Similarly, the **END LOOP** statement could be used to terminate the loop completely. In that case, execution would continue in the statement following the loop body (after the **UNTIL** statement). In this example, the index values from 1 to 4 will be printed, and the loop will be terminated even though the **DO..UNTIL** loop end condition has not yet been met. The value of the index variable in that case would be left at the last value (5, in the example above).

DOUBLE

The **DOUBLE** statement declares one or more variables as double precision floating point variables, and assigns them initial values. Optionally, the value can be declared as an array of double values by using an array size. For example,

```
DOUBLE X[10], SCALE=0.03
```

This statement creates an array with ten elements, each of type `DOUBLE`. The array is initialized to the default value for the type, so in this case it is an array of `0.0` values. The array size must be specified using square brackets, not parenthesis.

The statement also creates a single `DOUBLE` value named `SCALE`, which is initialized to a specific value of `0.03`.

This statement has the same effect of using the **DIM** statement with an explicit type of **DOUBLE**, but also allows you to specify an initial value for each item.

END

The **END** verb terminates execution of the current program, including programs that it has called as subroutines via the **CALL** statement or as functions.

```
END
```

For example, assume there is a program FOO which calls subprogram BAR which calls function COUNTER. If the function COUNTER executes an **END** statement, then this immediately terminates the execution of the function COUNTER and the programs BAR and FOO.

In additional to terminating execution, this statement also closes all files that were opened under user program control (the console is not closed, for example).

EXECUTE

The **EXECUTE** verb allows a string expression to be executed as a language statement. This allows a program to create a line of program code and have it processed by JBasic just as if it has been typed in by a user.

```
EXECUTE string-expression [ RETURNS variable ] [AS THREAD (NAME)]
```

If the statement contained in the result of the string expression gets an error, then the **EXECUTE** statement is considered to be in error and the program halts. You can use the **RETURNS** clause to capture the result of the statement in a variable; in this case the **EXECUTE** statement does not halt with an error, but the status information about the error from the statement is stored in the **RETURNS** variable.

The returned status can specify *scope qualifiers* to define the attributes of the variable. See the documentation on the **LET** statement for more information on scope qualifiers.

The status variable is a RECORD variable with the following fields:

Variable	Description
CODE	The error code (a mnemonic identifier) such as VERB
PARM	The argument that is included with the error, if any. For example, in a VERB error, it is the text of the unrecognized command.
SUCCESS	Boolean 'true' if this is a successful status code.

You can use the **MESSAGE()** function to convert this status record into a text message in the current language for display to the user if desired.

You can optionally cause the command to execute in another JBasic thread. A thread is an execution context; each JBasic thread is a new instance of JBasic that can execute concurrently - that is, program statements in each thread can execute at the same time. On computers with more than one CPU core, this means that jobs can be divided up to run on more than one processor at a time, reducing overall compute time. Alternatively, commands run on a separate thread can execute instructions that require waiting for an event such as user input while the primary thread continues executing.

```
EXECUTE "RUN PAYROLL_JOB" AS THREAD(TID)
```

This statement executes the command **RUN PAYROLL_JOB** on a separate thread. The thread is given a name, and this name is returned in the variable `TID`. You can use the **SHOW THREADS** command to see the state of execution of any threads that are created by your session. While the `PAYROLL_JOB` program is run, the current program or shell continues to run as well.

Because the command being executed might have come from user input, it is possible to constrain the possible operations of the command with the **SANDBOX** statement modifier.

```
INPUT “Command: “, CMD$
EXECUTE SANDBOX CMD$ RETURNS X
```

This example accepts a command (any JBasic statement) from the user and executes it. However, if the statement includes operations that might be destructive or interfere with the operation of the program, it is often desirable to limit the capabilities of the command.

The **SANDBOX** option indicates that the statement is run as if it was an unprivileged user. No file operations, **THREAD** operations, or operations that affect the stored programs are permitted in this environment.

FIELD

The **FIELD** statement defines the fields to be used to process a data record in a **BINARY** format file accessed via **OPEN**, **PUT**, and **GET** statements. The **FIELD** statement creates a named object that describes how to move data from the file to variables in memory and vice versa.

```
FIELD name, type [(size)] name [, type [(size)] name...]
```

Fields must have a valid name. This cannot be the name of any other variable or language keyword. This is followed by a list of one or more type and name combinations. The resulting field specification is then used in a **USING** clause in a **GET** or **PUT** statement to indicate how the binary data is to be processed. For example,

```
FIELD EMPREC, UNICODE STRING( 30 ) NAME, INTEGER(2) ID
PUT FILE BD, USING EMPREC, FROM { NAME : "Tom", ID : 2001 }
```

The first statement defines the record `EMPREC` which has two fields, a 30-character string suitable for storing Unicode (multi-national) strings called `NAME` and an short integer called `ID`. A maximum length is *always* required for a **STRING**, **UNICODE STRING**, or **VARYING STRING** field type. A length is optional for types of **INTEGER** and **FLOAT**. See the documentation on the **GET** and **PUT** statements for details on the supported binary file data types.

The **PUT** statement can be expressed with a **USING** clause that defines the field definition that describes what data is to be written to the file. The `EMPREC` tells the **PUT** statement to find fields named `NAME` and `ID` and write them out to the file from the **FROM** clause's data record.

The **FIELD** is a statement that defines a "map" of how data is to be read or written using **GET** and **PUT** statements. This is in contrast to *records* variable types *w*hich are a data type containing one ore more named fields in memory. The above syntax creates a **FIELD** that tells how to read and write *records*.

Instead of creating a field object, you can bind the field definition directly to a file. For example,

```
FIELD #1, UNICODE STRING( 30 ) AS NAME, INTEGER(2) AS ID
PUT #1
```

The first statement creates a field definition involving a string called `NAME` and a two-byte integer value called `ID`. These are associated with file #1, and each **GET**, **PUT**, or **SEEK** statement will use this definition to determine the fields to be read or written and how to calculate record positions in the file.

Each time the **FIELD** statement is executed, the record definition associated with the file changes, and subsequent **GET**, **PUT**, and **SEEK** operations will use the most recent **FIELD** definition.

The allowable types and characteristics for binary data types are shown in the following table:

TYPE	Description
`INTEGER`	An integer value. The default is a 4-byte integer in the range from -2147483648 to 2147483647. An optional size can be given of 1, 2, or 4 to specify byte, word, or integer values.
`FLOAT`	A floating point number. The default is a 4-byte single precision value, but a size can be given of 4 or 8 to select single or double precision data.
`DOUBLE`	A double-precision floating point number (8 bytes). This is the same as specifying FLOAT(8).
`STRING`	A string of text, with a specific length allocated in the record as specified by the size of the field.
`UNICODE STRING`	A string of Unicode (UTF-16) text, with a specific length allo-cated in the record specified by the size of the field.
`VARYING STRING`	A string of text of varying length. The maximum amount of space the string can take is specified in the SIZE field. An additional integer value is read or written that contains the actual length of the string in the buffer.
`BOOLEAN`	A single byte of data containing 1 or 0 to represent true or false values.

FILES

The `FILES` command lists files in the current directory, or in a directory given in the command:

```
FILES ["path"]
```

If omitted, the default directory is assumed. If a specific directory is to be listed, then that must be expressed as a string constant, including quotation marks.

More than one path can be specified on the command line. For example,

```
FILES "bin", "src"
```

This displays the files in the "bin" and "src" subdirectories.

FIND

The **`FIND`** command searches the current program for a specific string, and lists all the program lines that contain the string.

```
FIND keyword

FIND "string"
```

In the first form of the command, the current program is searched for any instance of the given keyword or numeric constant. Because the keyword is not in quotes, then only unquoted strings or capitalized keywords will match.

The second form searches for an exact match of the given string, including case. So the command **`FIND "instance"`** would locate the string inside a quoted text string or comment, but would not find the variable `INSTANCE` which would have been automatically capitalized in the program.

If there is no current program, then **`FIND`** reports an error. Use the **`OLD`** command to make an existing program current, or use the **`NEW`** command to create a new program.

If there are no copies of the selected string, then no error is reported but the message "`Found 0 instances`" is printed. If there are instances, each line of the program containing the search value is printed, followed by a count of the total number of instances there are of the search string in the current program.

FOR

The **FOR** statement creates a loop with an index variable that changes value by a specified incremental value each iteration of the loop. The loop executes as many times as it takes to increment the index variable past a given limit.

```
FOR index = start TO end [ BY increment ]
     ... statements ...
NEXT index
```

The *index* is a variable that is initially assigned the *start* value. The statements are executed, and when the **NEXT** statement is executed, the *index* variable has the *increment* value added to it. If the increment was not specified, then 1.0 is assumed. The index is then compared to the *end* value, and if the end value has been exceeded (greater than it if the increment is positive; less than it if the increment is negative) then the program continues executing with the next statement after the **NEXT**. Otherwise, the statement body is executed again.

If the index value is already past the limit, the loop does not run at all, but execution continues following the **NEXT** statement. An increment with a value of zero signals an error.

Loop statements such as **FOR...NEXT** cannot be executed directly from the command line, but are only valid in running programs.

An alternate version of the **FOR** statement allows you to specify a list of values to be assigned to the index variable rather than an incremented value:

```
NAMES = ["TOM", "MARY", "SUE", "BOB"]
FOR NAME = NAMES, "DAVE"
    PRINT "NAME = "; NAME
NEXT NAME
```

In this example, the index variable `NAME` will be set to the value of each item in the list that follows the "=" character. There is no **TO** or **BY** value. When one of the items in the list is an array, the index variable is set to each value of the array in turn. So in the above case, the index variable will be set to "TOM", "MARY", "SUE", "BOB", and "DAVE" in turn as the loop body executes. The values list can be a list of one or more expressions, which must be separated by commas.

Finally, you can use either form of the **FOR** statement to execute a single statement as the loop body with an implied **NEXT**. This is done by using the **DO** clause on the **FOR** statement:

```
FOR N = SYS$PROGRAMS DO PRINT "PROGRAM = "; N
```

This executes the **PRINT** statement for each value of `N` in the `SYS$PROGRAMS` array.

You can use the **CONTINUE LOOP** and **END LOOP** statements to change flow of control while executing the body of the loop. Consider the following example:

```
FOR I = 1 TO 10
    IF I = 5 THEN CONTINUE LOOP
    PRINT “Value “; I
NEXT I
```

In this case, when the value of the index variable is 5, the **CONTINUE LOOP** statement causes control to transfer to the **NEXT** statement, which iterates the loop. That is, it discontinues the current instance of the loop body, and runs the next loop iteration. If the ending condition is met, then the loop exits normally. In the above example, all the index values from 1 to 10 will be printed except the value of 5, because the **CONTINUE LOOP** causes the rest of the loop body to be ignored.

```
FOR I = 1 TO 10
    IF I = 5 THEN END LOOP
    PRINT “Value “; I
NEXT I
```

Similarly, the **END LOOP** statement could be used to terminate the loop completely. In that case, execution would continue in the statement following the **NEXT** statement. In this example, the index values from 1 to 4 will be printed, and then the loop will be terminated even though the **FOR** statement end condition has not yet been met.The value of the index variable in that case would be left at the last value (5, in the example above).

FUNCTION

The **`FUNCTION`** statement in a stored program declares a function, which takes zero or more arguments and returns a single value.

```
FUNCTION name ( [type] arg [, [type] arg...] )
```

The function is later called by the given name. The caller must supply at least as many arguments as are declared in the function. If there are no arguments, use empty parenthesis. If there are to be variable numbers of arguments, end the argument list with an ellipsis ("...") indicator.

By default, the function does not have to declare a type for what it returns. Rather, the **`RETURN`** statement is used in the function to pass a result back to the caller, and the function result is whatever type the value is. You can specify a specific data type for the function value, in which case the RETURN statement value is always converted to the required type. For example,

```
FUNCTION ARCTANGENT( X ) RETURNS DOUBLE
```

In this case, no matter what the ARCTANGENT program code calculates as a result (perhaps an integer or a double), the result will always be converted to a double before returning the result to the calling expression. The type of the parameters can also be optionally specified, as shown in this example:

```
FUNCTION FOO( INTEGER X, DOUBLE Y )
```

In this example, the parameters have explicit types given to them. When the type is given, whatever data is passed to the function is converted to the given type before being stored in the local argument variable. So the above example guarantees that the first value will be an integer, and the second will be a double, regardless of the values passed to the function. If the type names are not given, then the argument variables take on whatever type the passed parameter was in the calling program.

You can specific default values for parameters as well.

```
FUNCTION BAR( INTEGER X, INTEGER COUNT=10 )
```

In this example, the first parameter must be specified, and is converted to an integer. The second parameter may be omitted in the calling program; if it is not given then a value of 10 is assumed for the second parameter.

The function can be included anyplace an expression exists, and there is no distinction between stored program functions and intrinsic functions built into JBasic as far as how they can be used or called. See the section on Functions elsewhere in this document for a list of the built-in functions. Use the **`SHOW FUNCTIONS`** command for a runtime list of the built-in and stored program functions.

You can define additional characteristics of the function you create by using the **`DEFINE()`** clause following the function argument list. See the documentation on the **`PROGRAM`** statement for more details.

GET

The **GET** command is used to read information from a **BINARY** format file, which means the format of the data stored on the file matches the internal binary representation of the information in the computer, as opposed to a human-readable format. The information read from the file is stored in variables just like a **INPUT** statement that reads from a file. Data read from a **BINARY** file must be identified not only by the variable that will contain the information, but the type of data that the information is stored as on the disk file.

The binary format is specified in one of two ways: explicitly in the **GET** statement or via an array of record data types that describes what is to be written.

```
GET FILE BD, INTEGER ID, STRING(30) N, DOUBLE SALARY
```

In this example, the variables `ID`, `N`, and `SALARY` are read from the file. The variable `ID` is read as an integer, with a possible value in the range of +/- 2^31. The variable `N` is read as a string, with information on the maximum amount of space reserved in the file for 30 characters of data. The variable `SALARY` is read as a double precision floating point value.

Note that while the variable `N` may be any length from zero to 30 characters, space is reserved in the file for 30 characters of data. This will be important when the program needs to position itself in the file and must calculate the location of the binary data. See the documentation on the `SEEK` command for more information.

You can use the **FIELD** statement to create a definition of the contents of each data record in the file. This is then automatically used when a **GET** statement reads data. For example,

```
FIELD #1, STRING(30) AS NAME, INTEGER AS ID
GET #1
```

The **GET** statement reads a record from the file and stores the values in the variables **NAME** and **ID**. The **FIELD** stays in effect until another **FIELD** statement creates a new definition or a **CLEAR FIELD** statement removes the field definition completely. This field definition is used for both **GET** and **PUT** statements. See the documentation on the **FIELD** statement for more information.

If you use a FIELD that is not bound to a specific file, you can still reference it from a GET or PUT statement with the USING clause:

```
FIELD MYDATA, STRING(30) AS NAME, INTEGER AS ID
GET #1, USING MYDATA
```

GOSUB

The GOSUB statement transfers control to another statement in the current program, identified by a label. Unlike **GOTO**, when the program executes a **RETURN** statement, control will return to the statement following the **GOSUB**. This allows internal subroutines (without parameters) to be written in the current program.

```
        X = 3
        GOSUB TWOX
        PRINT X
        RETURN
TWOX:   X = X * 2
        RETURN
```

The above program will cause X to be set to 6. The **GOSUB** statement transfers control to a line within the same program that executes a **RETURN** when it is ready to return control to the program statement that called it.

In the above case, the subroutine TWOX could be called from many places in the program, and each time it would execute the same lines of code at the label TWOX and then return to the statement following whichever **GOSUB** directed it to transfer control.

This allows for simple subroutines to be implemented within the program for performing commonly performed operations that can be called from many places in the same program. When the subroutine is running, it has access to all the same symbols as the rest of the program, and any variable created or changed in the program will be in the same scope as the rest of the running program.

Obviously, most subroutines perform a more interesting operation than just doubling the value of a given variable. However, the important thing to note is that the subroutine does not have any additional information that is given to it other than the currently available variables. See the **CALL** statement for information on how to pass control to an entire program, giving it specific values for variables to operate on.

See the documentation on the **SUB** statement for information on how to create local subroutines in the current program that accept parameters and have their own variable scope.

GOTO

The **GOTO** statement transfers control to another statement in the current program, usually identified by a label.

```
    GOTO name
        or
    GOTO USING( expression )
```

The destination label can be identified by a name such as `TRY_AGAIN` or by a **USING** clause with an expression that resolves to a string, which is the name of the label. The destination statement has a label which is a name followed by a single colon, optionally followed by the first statement executed at that label. For example,

```
    GOTO TRY_AGAIN
     ...

  TRY_AGAIN:
     PRINT "We are going to try again"
     ...
```

For compatibility with older dialects of BASIC you can also use a line number as the destination of a GOTO statement. The program must have line numbers (as opposed to unnumbered statements in a workspace file).

```
1000  GOTO 1035
      ...
1035  PRINT “I AM HERE”
```

In this example, the statement at line 1000 will transfer control to the statement with line number 1035. A **RENUMBER** command will automatically update line number references in statements like GOTO that transfer control to a specific line in the program.

NOTE

Use of line numbers as branch destinations is not recommended because they are hard to remember and can change. Use text labels instead.

HELP

The **HELP** command displays information about commands, organized by topics which are one or more paragraphs of reference information and examples. If you just type **HELP**, you get an introductory topic that gives you information on other **HELP** items that are available.

- To see a list of help topics, use **HELP TOPICS.**
- To see a tutorial on entering and running programs, use **HELP PROGRAMS**
- To see information about available functions, use **HELP FUNCTIONS**

The text that is used for the **HELP** command output is part of the JBasic program archive file, jbasic.jar, and is in the member "JBASIC-HELP.TXT". You can edit this text if you wish to add additional information to the **HELP** command output.

The HELP command is a good example of a JBasic verb. The **HELP** command is implemented as a JBasic program that acts as if it was a statement in the language. You can see the source code for the **HELP** command by using the command **SHOW VERB HELP** once the program is loaded into memory by executing at least one **HELP** command.

IF

The **IF** statement conditionally executes one or more statement. The **IF** statement evaluates an expression whose result is tested to see if it is a true or non-zero.

```
IF X>3 THEN PRINT "We win!"
```

If the value of X is greater than 3, then the **PRINT** statement is executed. If X is less than or equal to three, then the next statement in the program is executed instead.

You can have **IF** statement include a statement to be executed if the condition is false.

```
IF X = 3 THEN PRINT "winner" ELSE PRINT "loser"
```

In this case, if X is 3 then "winner" is printed. If X is not 3, then "loser" is printed. If the **THEN** clause uses a compound statement (multiple statements separated by colons) then do not put a colon before the **ELSE** keyword. For example,

```
  IF X THEN PRINT “WINNER!” : GOTO DONE ELSE PRINT “STILL NO WINNER.”
```

Note that **IF** statements can contain **GOTO** statements that will effect flow of control in the program. The **IF..THEN..ELSE** statement above could have been written as

```
  IF X = 3 THEN GOTO WINNER
  PRINT "loser"
  GOTO DONE

WINNER:
  PRINT "winner"

DONE:
```

This code segment will use **GOTO** to conditionally execute a block of code.

For compatibility with older versions of BASIC, you can specify a line number to branch to if the expression evaluates to true or non-zero. For example, consider the following code fragment:

```
1000  IF X > 3 THEN 1035
...
1035 PRINT “X was greater than 3!”
```

The line number follows the **THEN** clause without a **GOTO** keyword. There cannot be an **ELSE** clause in this variation of the **IF** statement.

NOTE

When a **RENUMBER** command is given with the sample code above, if statement `1035` gets a new line number, then the statement at line `1000` will be modified as well to point to the new line number.

Use of line numbers to specific the destination of a branch operation is not recommended. The line numbers are not inherently meaningful to the programmer compared to a text label (**GOTO** `HANDLE_INPUT` means more to the programmer than **GOTO** `3310`). Additionally, because the **RENUMBER** statement changes the line numbering, the programmer will find that even choosing “easy to remember numbers” like `3000` will result in unpredictable changes to the line numbering scheme.

Finally, you can construct **IF**..**THEN**..**ELSE** logic using multiple lines of code. This requires that there be no statement text following the **THEN** keyword; the block of code executed must start on the following line. For example,

```
IF SALARY > 12.00 THEN
   WAGE = HOURS * SALARY           // Execute if condition true
   PRINT “STRAIGHT SALARY”
ELSE
   WAGE = HOURS * (SALARY * 1.20)  // Execute if condition false
   PRINT “ADJUSTED SALARY”
END IF
```

In this example, the *true* block of code will be executed when the condition is true (the value of `SALARY` is greater than 12.00). This block of code consists of all the statements before the **ELSE** statement, which must stand by itself. The *false* block of code is executed if the condition is false; all statements between the **ELSE** and the **END IF** will be executed in this case. The **ELSE** clause is optional, if there is no *false* block of code then the **THEN** clause is terminated with the **END IF** statement.

INFORMAT

The **INFORMAT** statement is used to compile a input format specification and store it in a record variable. This format specification can then be used with the **INPUT()** function to process an input buffer according to the format specification.

```
INFORMAT fmtvar AS format-spec [, format-spec]
```

The *fmtvar* is an identifier indicating a variable that will be created to contain the format specification. The specification is an array of records, made up of one or more *format-spec* fields. Each format specification defines an input operation or a positioning operation. For example,

```
INFORMAT DOLLAR_FIELDS AS INTEGER(*), SKIP(“.”), INTEGER(*)
FIELD_VALUES = INPUT( DOLLAR_FIELDS, “123.45”)
```

The first statement creates a format specification called DOLLAR_FIELDS that contains three specifications. The first reads an integer of variable length, the second skips until it finds a “.” character, and the third specification reads another integer of variable length. The second statement applies this format to a string value, and returns an array of values. The resulting data in FIELD_VALUES will be an array of two items: the integer 123 and the integer 45, skipping the decimal point between them.

The following table describes the input format specifications supported.

Specification	Description
`INTEGER(size)`	Input an optional sign “+” or “-” followed by an integer value. If the size is given as varying (indicated by an asterisk) then the input specification reads characters until a non-integer character is found. If a size is given, then exactly that many characters are read, and the integer must be right-justified in the field.
`SKIP(count)`	Skips the input pointer ahead *count* characters in the input field. These characters are not used for data input.
`SKIP(“string”)`	Skips the input pointer ahead until the *string* has been found, and positions the input pointer immediately following the string value.
`POS(count)`	Moves the input pointer to the specific position in the string, where 1 represents the first character in the input buffer, 2 indicates the second character, etc.

INPUT

The **INPUT** statement is used to prompt the user for a value or list of values, and stores them in one or more variables.

```
INPUT [FILE identifier,] ["prompt",] identifier
```

If the **FILE** keyword is given followed by a file identifier, then the input is taken from the named file. Otherwise, the user's console is prompted. If the prompt string is not given, the string content of the variable `SYS$INPUT_PROMPT` is used, which defaults to "? ". You cannot specify a prompt string when a **FILE** identifier is used. The **INPUT** identifiers must follow the same rules as identifiers on the left side of the “=” in a **LET** statement.

The prompt text is printed on the console (if specified) and JBasic then reads text from either the console or file to store in the named identifiers. If more than one variable value is to be read at one time, you should separate them by commas or blanks in the input text.

The input variable can specify *scope qualifiers* to define the attributes of the variable. See the documentation on the **LET** statement for more information on scope qualifiers, such as marking the variable **READONLY** after the data is stored.

The input is one or more values separated by commas, blanks or end-of-line. These values are assigned in the same order as they are given to the identifiers in the **INPUT** statement. For example,

```
    BASIC> INPUT "Enter three scores: ", A, B, C

    Enter three scores: 33,55,13
```

The prompt string is displayed and the user enters three values. Upon completion of the statement, the variable `A` will contain 33, the variable `B` will contain 55, and the variable `C` will contain 13. If more values are given on the input than are needed to satisfy the **INPUT** statement variable list, the remaining values are processed with the next **INPUT** statement. So if an **INPUT** statement requires input for a single variable but two are given on the input line, a second **INPUT** statement won’t prompt for input but will just use the additional value from the first input. Any unused input values are discarded when the program completes execution.

You can specify arbitrarily-complex expressions representing the location in which the input value is to be stored. For example, this assumes an array of employee records and sets the value of the employee name in an element of the array:

```
    INPUT “Enter your name: “, EMPLOYEES[EMPID].NAME
```

In this example, the variable `EMPID` is used to locate a specific record in the `EMPLOYEES` array and then set the value to the member `NAME` in that record. If the variable references an array index position or record member does not exist, then it is created automatically.

You can specify the type of data that is to be returned in the input statement regardless of the type of data found in the file by giving an **`AS`** clause. For example,

```
INPUT #3, NAME AS STRING
```

This reads a value from the input file, and converts it to a **`STRING`** data type.

You can also specify the special data type of **`XML`** which means that the file is read until a valid XML string is found. The resulting string is processed just as if it was passed through the **`XMLPARSE()`** function and attempts to read a value from the XML string. If there is no valid XML data in the file then a `SYNTAX` error is generated. If the XML string is incomplete, an `EOF` error is generated.

```
INPUT #3, STUDENT_DATA AS XML
```

After the input file is read, the file is left positioned after the XML string. By default the root tag of the XML is *Value* (case insensitive). You can specify an explicit root tag by specifying it as a string expression in parenthesis, such as the following. The XML format must still be that of a JBasic *Value* but can have a custom root tag name:

```
INPUT #3, STUDENT_DATA AS XML("STUDENTDATA")
```

Finally, you can also specify a special data type of **`RAW XML`** which means that the file is read until a valid XML string is found, which is returned as a JBasic string. If there is no valid XML string in the file then a `SYNTAX` error is generated. If the XML string is incomplete, an `EOF` error is generated.

```
INPUT #3, USERDATA AS RAW XML
```

After the input file is read, the input is left positioned after the XML string. If valid XML was encountered in the file, then a complete XML specification will be found in the string variable `USERDATA` after the above statement executes. The is the responsibility of the program to determine how to use or process the XML data.

Another use of the **`INPUT`** statement is to allow the user's input to determine a label for each value, which is returned to the program as a record. This is done with the **`BY NAME`** clause. For example,

```
    BASIC> INPUT “Data? “, FOO BY NAME
    Data? User=”tom”, age=38
```

This prompts the user for input. The user must enter one or more comma-separated values with a variable followed by an equals sign and the value. Each user entry becomes a field in the resulting record. For example, `FOO` in the above case will be set to the value `{ AGE:38, USER:”Tom”}.`

Another version of this syntax allows the user’s input to directly set you specify in the **`INPUT`** statement using the **`BY NAME()`** clause. For example,

```
   INPUT "Parms? ", BY NAME( AGE, GENDER, NAME )
```

The user is prompted and enters a line of text consisting of a comma-seaprated list of the given names, an "=", and a value. The names can be given in any order in the input. The following is an example of an immediate mode **`INPUT`** statement showing the user’s input.

```
  BASIC> INPUT "Parms? ", BY NAME( AGE, GENDER, NAME )
  Parms?  GENDER="F", AGE=35, NAME="Sue"
```

When the statement completes, the three variables are given the values named by the user. If the user does not give a named value for a variable in the **`BY NAME()`** list, then that variable is set to a double missing value (NaN). If there is a syntax error in the user's input, all variables are set to NaNs. You can specify a different name for the user to specify than the variable it is assigned to, as in

```
  INPUT "Parms? ", BY NAME( PARM1 AS P1, PARM2 as P2)
```

In this case, the user input must reference `P1` and `P2`, but the values are stored in variables `PARM1` and `PARM2`. This allows you to specify simplified names for the user input.

If the user specifies values that are not in the **`BY NAME()`** list, they are normally ignored. However, you can capture a list of the unexpected variables using the **`UNEXPECTED AS`** clause:

```
     INPUT BY NAME(USER, PASSWORD) UNEXPECTED AS EXTRAS
```

If the user's input contains any values other than `PASSWORD` and `USER`, the names of the values are stored as record named `EXTRAS` If there were no unexpected values in the input then the record will be empty. Otherwise the record contains the names and values that were unexpectedly found in the input.

In any use of the **`INPUT BY NAME`** notation, if the user’s input is not syntactically valid, then the global variable `SYS$STATUS` is set to reflect the error detected, such as invalid numerical notation or missing “=” character, etc.

```
    TABLE EMPDATA, STRING NAME, INTEGER AGE, DOUBLE SALARY
    INPUT “Enter employee data: “, ROW OF EMPDATA
```

The **`INPUT`** statement can also be used to input a row of data from a table. The **`INPUT ROW OF`** statement processor uses the description of the table to determine the required types of data for the input stream. It reads as many items as needed from the input buffer to satisfy the requirements for one row of the table, and performs the type conversions as needed automatically. If a prompt is not given in the statement, then a prompt is constructed using the names of the columns. The new row is added to the end of the table.

INTEGER

The **INTEGER** statement declares one or more variables of type INTEGER, and assigns them initial values. Optionally, the value can be declared as an array of integer values by using an array size. For example,

```
    INTEGER X[10], AGE=49
```

This statement creates an array with ten elements, each of type `INTEGER`. The array is initialized to the default value for the type, so in this case it is an array of zeroes. The array size must be specified using square brackets.

The statement also creates a single `INTEGER` value named `AGE`, which is initialized to a specific value of `49`.

This statement has the same effect of using the **DIM** statement with an explicit type of **INTEGER**, but also allows you to specify an initial value for each item.

KILL

The **KILL** statement is used to delete a file, directory, or execution thread. In the case of a file or directory, the argument is a string containing the name of the path or file. If the argument is a string identifying a path to a directory (as opposed to a file), the directory must be empty or an error occurs.

```
KILL "names.txt"
```

The statement will signal an error if the file does not exist, or if security restrictions prevent the file from being deleted.

You can also **KILL** an open file by referencing it using its file handle, indicated by the **FILE** keyword:

```
KILL FILE MYFILE
```

The handle variable `MYFILE` must reference an open file. The file is closed and deleted; after this statement the file handle is no longer valid.

Additionally, you can use a form of the **KILL** statement to stop a running thread.

```
  TLIST = THREADS()
  KILL THREAD TLIST[1]
```

This uses the **THREADS()** function to get a list of the active threads, which is a list of strings. Each string contains the name of the instance of JBasic running the thread (the `SYS$INSTANCE_NAME` variable in that thread). This string is used with the **KILL THREAD** statement to stop that thread from running.

Note that a thread that is waiting for an *event* like user input or reading from an empty queue will not stop until the *event* is complete.

LET

The **LET** statement assigns a value to a variable. If the variable does not exist, it will be created. If the variable already exists and is *read-only*, an error is signaled

```
[LET] variable = expression
```

The verb **LET** does not have to be explicitly specified in the statement, though it will be added automatically by JBasic. By default, the variable's type is set to whatever type the expression result is, even if that is different than the variable's previous type. For example,

```
 X = 33
 X = "Test"
```

This is a valid sequence of statements, and after both statements are executed, the variable X will be of type *string*, and contain the characters "Test".

A variable can be created as an array, simply by specifying an array index in the left side of the expression. For example,

```
NAMES[3] = "Mary"
```

If the array NAMES does not exist, it is created, and the third entry is set to the string "Mary". The array elements NAMES[1] and NAMES[2] will be set to empty strings. Specifying an array subscript that is larger than the current arrays size results in the array being extended automatically.

You can specify arbitrarily complex expressions representing the location in which the expression is to be stored. For example, this assumes an array of employee records and sets the value of the employee name in an element of the array:

```
  EMPLOYEES[EMPID].NAME = “Tom”
```

In this example, the variable EMPID is used to locate a specific record in the EMPLOYEES array and then set the value to the member NAME in that record. When used in a **LET** statement (or as the target of an **INPUT**, **LINE INPUT**, or **READ** statement), if the array index position or record member does not exist, then it is created.

The **LET** statement can also define additional information about the variable using *scope qualifiers.* These are indicators that define information about what symbol table (such as the local, parent, root, etc.) the variable is created in, or if the variable is to be marked as **COMMON** or **READONLY**. These scope qualifiers are specified after the name of the variable, and can only be given after a scalar variable. That is, you can set the scope of the variable `X` in a **`LET`** statement, but you cannot apply scope to an element of a compound variable reference such as `X[I].KIND`.

A scope qualifier can be specified right after the name of the variable, separated by a "slash" character, as in this example:

```
X/PARENT/ = 3.5
Y/COMMON,READONLY/ = "Bubba"
```

In these examples, the variable `X` is created in the parent table - that is, the table of whatever program called the current program. In the second example, the variable `Y` is marked as both `COMMON` (which means it will be copied to any program invoked with a **`CHAIN`** command) and is also `READONLY` which means it cannot be modified once it is set. When more than one scope qualifier is used, they are separated by commas.

The permitted scope qualifier names are:

Qualifier	Description
`LOCAL`	The variable is created in the local (current) symbol table. This is the default behavior of any statement that creates a variable and does not need to be explicitly stated.
`PARENT`	The variable is created in the parent of the currently running statement. If the statement is in a program that was called from another program, the variable is created in that program's symbol table. If in a program run from the command prompt, the variable is created in the shell's symbol table and will be available after the program or statement terminates.
`ROOT`	The variable is created in the `ROOT` symbol table. The `ROOT` symbol table is the "ancestor" of all other symbol tables. A symbol that is stored in the `ROOT` table will be visible to any program or statement running in any thread. This is one way that information can be shared between running JBasic program threads.
`COMMON`	The variable is marked as a `COMMON` variable, which means that it will be copied to the symbol table space of any program that is invoked with a CHAIN statement from the current program.
`READONLY`	The variable is marked as `READONLY` which means that it cannot be modified once the value is set by the current statement.

Note that these scope qualifiers can also be set on any other statement that creates a variable value, such as **`INPUT`**, **`LINE INPUT`**, **`READ`**, **`GET`**, or **`DIM`**. An exception is that they cannot be used on the index variable of a **`FOR`** statement.

LINE INPUT

The **LINE INPUT** command reads a complete line of text from a file or the command-line console and stores it in a string variable. An optional prompt string can be given if input is from the console.

```
LINE INPUT [ FILE identifier ] [ "prompt text", ] name
```

The input is read from the console by default unless a **FILE** identifier clause is given. This must be the identifier for an open file; see the documentation on the **OPEN** statement for more information.

The prompt text can only be specified when reading from the console; it cannot be specified if the **FILE** clause is used. If input is from the console and no prompt string is given, then there is no prompt text output to the console.

The user can enter a single line of text with any internal punctuation, spacing, etc. and the entire line of text is stored as a string in the named variable. If the input is from a file, then a single line of the file is read in and stored in the string variable. This is different than the **INPUT** statement which will evaluate the text entered and look for multiple values to store in multiple variables. The **LINE INPUT** statement will always read an entire line of text, and store it in a single variable.

You can specify arbitrarily complex expressions representing the location in which the expression is to be stored. For example, this assumes an array of employee records and sets the value of the employee name in an element of the array:

```
LINE INPUT “Enter your name: “, EMPLOYEES[EMPID].NAME
```

In this example, the variable `EMPID` is used to locate a specific record in the `EMPLOYEES` array and then set the value to the member `NAME` in that record. When used in a **LINE INPUT** statement (or an **INPUT**, **LET**, or **READ** statement), if the array index position or record member does not exist, then it is created.

The input variable can specify *scope qualifiers* to define the attributes of the variable. See the documentation on the **LET** statement for more information on scope qualifiers.

LIST

The **LIST** command displays the current program's text to the console.

```
LIST
```

The program is output in a formatted fashion, as opposed to the exact spacing as originally entered. You can list a specific line, or a range of lines. If the beginning or ending of the range is not given, then the first or last lines are assumed.

```
LIST 1150                 // List line 1150
LIST 100-150              // List lines between 100 and 150
LIST -30                  // List lines up to line 30
LIST 500-                 // List lines after 500
```

You can use statement labels for line numbers as well. The label must exist or an error is reported.

```
LIST INIT - TERM       // List lines between INIT and
                       //    TERM statement labels.
```

The output can be controlled by several global variables, which affect the formatting of the program text.

Variable	Description
SYS$LABELWIDTH	This variable describes the default number of characters reserved for statement labels in the output. Statements with no label have this many blanks at the start of each line so that program statements line up and are easier to read. The default is 10 characters.
SYS$RETOKENIZE	This variable indicates if the program text entered by the user is retokenized into uniform format. When set, JBasic uses a set of internal rules to insert spacing in the program line and capitalization of non-quoted strings. The default is true.
SYS$SOURCE_LINE_LENGTH	Statements longer than this number of characters are "wrapped" by using the continuation character "\" at the end of the line, and subsequent characters appear on the next line. The default is 80.

LOAD

The **`LOAD`** statement loads a text file into memory as an executable program element. The text file can contain one or more program, function, verb, or test definitions.

```
LOAD "program-file"
```

If you specify a file name without an extension, the extension of "`.jbasic`" is assumed. The program(s) in the named file are added to the programs already in memory - multiple **`LOAD`** commands result in adding programs to memory - not replacing the ones already in the workspace - unless the file contains programs with the same name as something already in memory.

The format of a program file is an editable text file containing all the program elements to be loaded. Each program element must start with a program element declaration statement, such as **`PROGRAM`**, **`VERB`**, or **`FUNCTION`**. Programs may have line numbers, but they are optional. However, programs loaded from a workspace that do not have line numbers are given line numbers by default starting at 100.

All of the lines of text following a program element declaration statement are added to the program being stored, until the end of the file or the next program element declaration. This means that comments that appear before a **`PROGRAM`** statement will be stored in the previous **`PROGRAM`**, not the current program!

If the first element in the file does not contain a **`PROGRAM`**, **`VERB`**, or **`FUNCTION`** statement, then the file name is used to create a default **`PROGRAM`** name.

There is a special case of **`LOAD`**, when the file being loaded is an XML representation of a program. This kind of file can be created using the **`SAVE XML`** command. In this case, the file can contain only one instance of a program, and it must be correctly formed XML.

LOCK

The **LOCK** command is used to create a lock if it doesn't already exist, and put a hold on the lock so no other JBasic thread in the current program can access the lock. Specify one or more lock names in a comma-separated list:

```
LOCK COUNT_L, COUNT_Q
```

The example above creates locks COUNT_L and COUNT_Q (if they don't already exist) and then holds the locks. This will prevent any other thread from being able to acquire the same locks.

Another thread that executes a **LOCK** statement for the same named lock will wait until the lock is re-leased by the thread that holds it. The **UNLOCK** statement is used to release locks so other threads that need the locks can then hold the locks.

Locks are used to create a "critical region" of your programs that will prevent more than one concurrent thread of execution from performing conflicting operations. Consider the following sample code:

```
    LOCK COUNT_L
    COUNT = COUNT + 1
    UNLOCK COUNT_L
```

If this same code is executing simultaneously in more than one thread, there is a risk that the threads will interfere with each other. The first thread will load the value of COUNT, and add 1 to it. The second thread would then load the same value of COUNT and add 1 to it. The first thread would write the newly updated value back to COUNT, and then the second thread would write the *same* value back to COUNT. The result is that - depending on the order of instructions being executed simultaneously - both threads will attempt to increment COUNT but it will effectively only be incremented once.

By using the **LOCK** and **UNLOCK** statement in the example, each thread is guaranteed to have exclusive execution control while they hold the same lock being shared among multiple threads, and the increment of COUNT will be completed on one thread before it can run on another thread.

When a program has created a dynamic lock name, where the name is stored in a string variable, the **LOCK** statement will accept a **USING** clause, as in:

```
    LOCK USING( “LOCK_” + LCKID )
```

This example assumes that an integer value LCKID will be appended to the string "LOCK_" to indicate the specific lock being manipulated.

You can create a lock without acquiring access to it using the LOCK CREATE command. For example,

```
LOCK CREATE MYLOCK

      or

LOCK CREATE USING (“LOCK_” + LOCKID )
```

In these examples, a new lock is created and owned by the current process. However, the lock is not currently in a locked state, as indicated by a **SHOW LOCK** command, or by using the `LOCKS()` function to return an array of records describing each active lock.

See the documentation on the **UNLOCK** statement for more information. You can use the **SHOW LOCKS** command to display the list of known locks in the JBasic session. You can use the **CLEAR LOCK** command to delete the locks, and you can use the **LOCKS()** function to get an array of records describing the locks.

Note that locks are global to a process; all sessions and all threads in a single process will have access to the same table of locks and lock names.

MESSAGE

The **MESSAGE** statement defines a signal name and the text used to format the associated message in a given language.

```
MESSAGE identifier (language) "format string"
```

The identifier must be a valid JBasic name that is not already a reserved word, function name, or statement name. This is followed by a language code, which must be a two-letter language code that corresponds to the user’s language. `EN` is the default, and means English. Other possible codes include `FR` for French or `ES` for Spanish. These codes are defined by the Java standards for language encoding.

The format string is the text string used to display the signal when it is formatted for output as an error message. If the signal allows an argument, it can be placed in the format string using “[]” as s a placeholder.

For example,

```
MESSAGE VERB (EN) "Unrecognized command, []"
MESSAGE VERB (ES) “Comando desconocido, []”
```

This defines the same message code with two language encodings (English and Spanish). The text that will be displayed in the event of a `VERB` error will be selected based on the user’s current language setting as a user, typically a setting of the operating system such as the Mac OS X “System Preferences” panel.

You can override the setting of the language by changing the JBasic variable named `SYS$LANGUAGE`, which must be a two-character string identifying the language encoding.

MID$

The **`MID$`** pseudo-function is used on the left hand of an assignment operation to store character data into the middle of an existing string, without disturbing other characters in the string.

```
NAME = "I AM BOB SMITH"
MID$( name, 6, 8 ) = "TOM"
PRINT NAME
```

The above sequence will print out "`I AM TOM SMITH`", because characters 6-8 of the named string variable were replaced with the string expression "TOM".

The string variable used in the **`MID$()`** pseudo-function must already exist; you cannot create a new variable using the **`MID$()`** operation.

The position within the string is 1-based; that is, the first character is position 1, and the last character is at the position also returned by the **`LENGTH()`** function. A runtime error is generated if the start position is less than 1, the end position is greater than the length of the string, or if the end position is less than the start position.

If the string value to be inserted is shortened than the required number of characters it is blank-padded. If it is longer than the available space, the string is truncated such that only the left-most characters are used.

Note that *scope qualifiers* are **not** permitted in the **`MID$`** pseudo-assignment operation. See the documentation on the **`LET`** statement for additional information on scope qualifiers.

MULTIPLY

The **MULTIPLY** statement multiplies an existing variable by the value of an expression.

```
    MULTIPLY HOURS BY RATE
```

The above statement calculates the expression and adds it to the existing value of `HOURS`. If the variable `HOURS` does not exist, then this statement generates a runtime error. This is essentially equivalent to the statement **LET** `HOURS = HOURS * RATE).`

Note that if this expression uses pre- or post-increment or decrement operators in the target expression, they are processed after the statement completes the storage of the new value in the target. For example,

```
   B = 3
   MULTIPLY EMPS[B++] BY X+B
```

The increment operation of `B` will take place after the sum `X+B` is multiplied by the array value `EMPS` at the location of the current value of `B` (3 in this case).

The **MULTIPLY** statement can also be used to repeat a string value:

```
   X = "-"
   MULTIPLY X BY 80
```

The result is that the variable will contain a string containing 80 dashes. The value to be multiplied must be a string and the value it is multiplied by must be an integer.

NEW

The **NEW** statement creates a new program with no statements in it. You enter new statements by entering commands that are preceded with a line number. Statements may be entered in any order, bur are stored in line-number-order.

```
NEW [kind] [name] [arguments]
```

If you do not specify a kind of **FUNCTION**, **VERB**, or **TEST**, then **PROGRAM** is assumed. If you do not give the program a name, it will be given a unique name. This is the name that the program will be referenced by in a **SHOW PROGRAMS** listing or when you wish to call the program from some other program. The argument list is optional and can be used with **PROGRAM** or **FUNCTION** programs only.

The program is created with a declaration statement, a few header comments, and a **RETURN** statement. You can use the **RENUMBER** command to change the line number ordering in the program as you add new statements.

Here is an example of creating a new **FUNCTION** named DBL, and then entering a statement to represent the body of the function. In this case, the function is defined with an argument list, and that can be put on the **NEW** statement. If no arguments are provided, then the function won't accept arguments until the **FUNCTION** statement is subsequently modified. See the documentation at the introduction of this manual on creating programs for more information.

```
NEW FUNCTION DBL(X)
1040 return x*2
```

You can create a new program from a record value. For example, if you use the **PROGRAM()** function to capture the contents of a current program, the resulting record can be used to construct a new instance of the program:

```
NEW USING( MYPGM )
```

In this example, the variable MYPGM must contain a record with members for NAME (the name of the program), USER (a boolean indicating if it is a user program or not), and LINES (an array of strings containing the individual program statements). The result is the creation of a new program with the given name and program lines. The following example effectively copies "OLDPGM" and creates a new program called "NEWPGM" using the program data:

```
OLDPGM = PROGRAM(“OLDPGM”)
OLDPGM.NAME = “NEWPGM”
NEW USING( OLDPGM )
```

NEXT

The **`NEXT`** statement identifies the end of a **`FOR...NEXT`** loop construction. See the documentation for the **`FOR`** statement for more information.

Loop statements such as **`FOR...NEXT`** cannot be executed directly from the command line, but are only valid in running programs.

OLD

The **OLD** statement makes an existing program become the current program. The program name must exist as a program in the stored program memory. See the **LOAD** and **SAVE** commands for information on how to bring programs into and out of stored program memory from other locations like a disk.

```
OLD name
```

The existing program (if any) is not disturbed, but the program named in the **OLD** statement becomes "current". That means that a **RUN** command with no argument will run that program, and a **LIST** command will list it. Statements can be stored in the program by entering them with a line number on the command line.

You can use **OLD** to access the program text for non-program objects by including the type.

```
OLD FUNCTION MIXEDCASE
```

This makes the function `MIXEDCASE` be the current program object that can be modified or viewed (via **LIST**). You cannot access intrinsic functions this way, since they do not have associated JBasic program text.

ON

The **ON** statement declares a statement label to be executed in the event of an error being signaled. The error could be generated by the program code itself (see the **SIGNAL** statement) or could be the result of an error in the program syntax or execution signaled by JBasic. For example,

```
ON ERROR THEN GOTO ERROR_HANDLER
```

This will cause control to transfer to the label `ERROR_HANDLER` when any error occurs.

```
ON TOOBIG THEN GOTO FIX_UP
```

This will cause a **GOTO** to the routine `FIX_UP` to be executed if the specific error `TOOBIG` is signaled.

An **ON** statement only applies to the program it is executed in. For example, if **PROGRAM** `FOO` calls **PROGRAM** `BAR` using a **CALL** statement, and **PROGRAM** `BAR` has an **ON** statement, then only errors executed while in **PROGRAM** `BAR` will trigger the **GOTO** statement. When **PROGRAM** `BAR` executes a `RETURN` statement, then the **ON..THEN GOTO** is discarded. Each **PROGRAM** may have its own **ON** statements, so **PROGRAM** `BAR` and **PROGRAM** `FOO` may each have an **ON ERROR** statement, and each is executed when the error is signaled in the program that contains it.

An **ON** statement need only be executed one time in the program to stay in effect for as long as that program is running regardless of whether an error is triggered or not.

When an **ON..THEN GOTO** statement is executed, the `SYS$STATUS` record variable describes the specific error.

Variable	Description
`SYS$STATUS.CODE`	The error code (a mnemonic identifier) such as VERB
`SYS$STATUS.PARM`	The argument that is included with the error, if any. For example, in a VERB error, it is the text of the unrecognized command.
`SYS$STATUS.SUCCESS`	Boolean 'true' if this is a successful status code.
`SYS$STATUS.PROGRAM`	The name of the program running when the error occurred.
`SYS$STATUS.LINE`	The line number of the program when the error occurred.

OPEN

The **`OPEN`** statement opens a file for data access. The file can then be used with a **`PRINT`**, **`LINE INPUT`**, or **`INPUT`** statement if it is a text file, or **`GET`**, **`PUT`**, and **`SEEK`** statements if it is a binary data file. You must specify the path name of the file, the mode (**`INPUT, OUTPUT,`** etc. from the table below) and provide an identifier used to reference this file later.

```
OPEN FILE "full-path-name" FOR mode AS identifier
```

Any of the statement clauses **`FILE`**, **`FOR`**, or **`AS`** can appear in any order, but all must be specified. The identifier is used to locate this specific file later for statements that reference the file such as a **`PRINT`** statement.

The *mode* must be one of the following keywords:

Mode	Description
`INPUT`	The file must already exist and is opened for input.
`OUTPUT`	The file is created if needed and is written to from the beginning of the file.
`APPEND`	The file is created if it does not exist. Writing begins at the end of the existing file if there is any existing content.
`BINARY`	The file is created as a "binary" file which means that numbers and strings are stored in the file just as they are stored in computer memory.
`QUEUE`	The file is an in-memory thread-safe FIFO (first-in, first-out) queue used to communicate messages between JBasic threads.
`PIPE`	The file is a command to the local host system which is executed by JBasic. The output of the command becomes the "contents" of the file and can be read using **`LINE INPUT`**, etc.
`DATABASE`	The file is a JDBC database, and the file path name is a DSN record instead of a string variable. See the section on **Databases** in the first chapter of this document for more information.

By default, the file is a "text" file, meaning all output is written as human-readable text. A file created using **`INPUT`**, **`OUTPUT`**, or **`APPEND`** modes can be opened by a text editor or printed to a printer.

By comparison, a **`BINARY`** file has its information stored in the internal representation of the data as used by the computer itself, and is generally not readable by a person or usable in a text editor. However, **`BINARY`** files have the benefit of being "random access." This means that the program can decide exactly where in the file (using a relative byte position) a data item is to be read or written. This lets the JBasic program organize the data in the file into "records" that are collections of like data, such as name, age, and salary, and access records directly (rather than having to read all records to find a specific one.) See the documentation on the `GET`, `PUT`, and `SEEK` statements for more information.

For compatibility with other dialects of BASIC, there is considerable flexibility in the expression of the `OPEN` statement clauses. The clauses are defined as the **`FILE`** clause which names the path to the file, the **`FOR`** clause which names the mode of access, and the **`AS`** clause which tells the file identifier to use for subsequent access to the file. These clauses may appear in any order, and the keywords **`FOR`** and **`FILE`** are optional. As an example, the following statements are equivalent:

```
OPEN FILE “X.DAT” FOR OUTPUT AS MYFILE

OPEN OUTPUT FILE “X.DAT” AS MYFILE

OPEN AS MYFILE “X.DAT” FOR OUTPUT
```

Additionally, file identifiers can be specified in one of two ways. The default mode for JBasic (and the one used in most examples in this document) is that file identifiers are names, just like variable names. In the above examples, `MYFILE` is the identifier which would be used on subsequent **`PRINT`** or **`CLOSE`** statements. Some dialects of BASIC require that files be identified by an integer expression instead of an identifier. In this case, the integer expression must be preceded by a pound-sign (“#”) character.

```
OPEN “X.DAT” FOR OUTPUT AS #3
```

In this case, all references in the program to this file must be made as #3 rather than using the **`FILE`** keyword. So to print a line of text to this file, the statement would look like this:

```
PRINT #3, “This is a test line of text”
```

When numeric file notation is used, the file number is converted to an identifier, so a **`SHOW FILES`** command will show something like __`FILE3` for the open file.

For **`OUTPUT`** text files, you can specific automatic column formatting by adding the **`COLUMNS`** clause. This lets JBasic assist you in creating columnar output. For example,

```
OPEN FILE "x.txt" FOR OUTPUT AS X COLUMNS(25,3)
```

This means that each **`PRINT`** statement to the file that does not include a newline will be aligned into 25-character-wide columns, and every three columns a newline is automatically inserted. If you then **`PRINT`** to the file with a newline, then the column position is reset to the first column but formatting continues. Note that if you do not use the trailing ";" to prevent a newline character, then column formatting is not used.

In the column specification, if the width is less than zero, it means that the column is right-justified. If the width is greater than zero, the column is left justified.

As documented in the introductory section on threads, you can also create a reference to a queue, which is like a list of strings. The first string written to the list will be the first one removed, but at any given time more strings may be on the list than there are readers ready to remove them. A queue is created by using the **QUEUE** keyword.

```
OPEN QUEUE "TASK_LIST" AS MYQ
```

This creates the queue `TASK_LIST` if it does not already exist. A queue resides only in memory for the current process, and never is written to disk. If the queue already exists, then it is opened. Unlike text files, a queue can be read and written to using the same file reference. Once a queue is opened, you can use **PRINT** and **LINE INPUT** statements to write strings to the queue or read them back in. This is usually used between threads, but can also be used within a main program.

Because the queue is referenced as an open file, you will see it in the list of open files when you issue a **SHOW FILES** command. However, there is only one set of queues for the entire JBasic process, so there may be queues that were created by other threads that do not appear as open files in the main thread or a thread that issues a **SHOW FILES** command. In this case, you can use the **SHOW QUEUES** command to see all active queues on all threads. When the last thread closes a queue, it is deleted.

Another special file type is a **PIPE**, which is really a connection between JBasic and another program being run on the same computer system. The file name is the command that invokes the program. When the program runs, it normally produces output. This output becomes the "contents" of the file, and is read using conventional JBasic statements such as **LINE INPUT**. Here is a simple example of a program that uses a **PIPE** file type to read the output of a UNIX directory listing command.

```
OPEN PIPE "ls -l" AS #1
DO WHILE NOT EOF(1)
   LINE INPUT #1, TEXT$
   PRINT TEXT$
LOOP
CLOSE #1
```

This approximates the behavior of the command **SYSTEM "ls -l"** which also executes the command and displays the output. Of course, the advantage of the PIPE is that your program can use the output in the JBasic program rather than just displaying it on the console as in the above example.

Normally, a **PIPE** is similar to an **INPUT** file in that you read the contents of the file (the program output) and process it in your program. You can also treat a **PIPE** as an **OUTPUT** file if the program you run is expecting input; you can then **PRINT** information that is sent to the program being run. In general, it is safe to use a **PIPE** for input or output. If you write a program that attempts to perform both operations, be aware that you can *deadlock* your program. This means that if you run a command that expects input, and then you issue a **LINE INPUT** statement to wait for input, then both your program and the **PIPE** program will wait forever, each expecting the other to provide data.

Finally, you can use a variation of the **`OPEN`** statement syntax that emulates the behavior of programs written for GW-Basic or similar dialects. For example,

```
OPEN “FOO.TXT”, “I”, 3
```

This opens the text file “FOO.TXT” for input as file #3. The mode must be either “`INPUT`”, “`OUTPUT`”, or “`APPEND`”, or the first letter of those words. The file name can be any string expression.

The above statement has the exact same effect as the JBasic dialect version:

```
OPEN FILE “FOO.TXT” FOR INPUT AS #3
```

PRINT

The **PRINT** command prints output to the console or a file. The **PRINT** statement accepts an optional **FILE** clause, followed by a list of one or more expressions. Each expression is printed in order, followed by an end-of-line character.

```
   PRINT [ FILE identifier, ]  exp [, exp [, exp...]]
```

If a **FILE** clause is not given, then the output is directed to the user's console. If a **FILE** statement is given, then the output is written to the file referenced by the *identifier.* See the documentation on the OPEN statement for information on how this identifier is created. If a **FILE** clause is given, then the file must have been opened for **OUTPUT** or **APPEND** modes.

If the expressions are separated by a comma character "," then a tab is printed between each character so they are spaced in columns. If a semicolon ";" is used to separate the elements, then they are printed directly next to one another. You can include a trailing comma or semicolon so that output does not go to a new line after the **PRINT** statement output is generated.

If a **PRINT** statement is given without a list of expressions, then a newline is automatically added to the output (either the console or the named output file).

The **PRINT USING** statement is an extension of the **PRINT** statement that supports formatted text output. A single format string expression is given, followed by a list of comma-separated data items that are used to fill values into the format string. Unlike the standard **PRINT** statement, all elements in the list must be separated by commas.

```
  PRINT USING "Credit  ##.##     Net (###.##)", 3.5, -27.55

  Credit   3.50     Net ( 27.55)
```

The above example prints two values using a format string. The format options are the same as those used in the **FORMAT()** function. Text in the format string not part of the value format itself are just printed. There must be as many values in the variable list as there are in the format string expression itself.

You can print the output to a file as well, similar to a standard **PRINT** statement:

```
    PRINT FILE MYFILE, USING "000#.##", 1.3
```

The above example will result in the string "0001.30" being printed to the file opened as identifier MYFILE.

In a **PRINT USING** statement, you can use a trailing comma or semicolon to prevent a newline from being added to the output if you wish to print additional information on the same line of the output or file.

For compatibility with older dialects of BASIC, you can use the *question mark* (“?”) symbol instead of a **PRINT** statement on the command line. This is short-hand for asking “what is”, as in “what is X times 5?” as shown in this example:

```
BASIC> X=3
BASIC> ? X*5
15
BASIC>
```

You can also specify that the name of the variable be printed when outputting a given value. This creates self-descriptive output, as in

```
BASIC> PRINT X=, Y=
X=15        Y=”Age”
BASIC>
```

In this example, the name of the variable X is printed followed by an equals sign, and the current value. The formatting operator (the “,” comma) means a tab stop is skipped, and then the second variable name is printed followed by “=” and the value. This syntax can only be used for simple variable names; it cannot be used for array or record notation or an error is generated.

PROGRAM

The **PROGRAM** statement appears in a text file and identifies a block of statements to be stored as an executable program in the JBasic runtime.

```
PROGRAM BOB
```

This defines a program that will be named `BOB`. This must be the first line of a new program that you create. In fact, the **NEW** command will automatically create a first line containing a **PROGRAM** statement for you.

If you modify the **PROGRAM** statement in the current program, it effectively renames the current program. You will immediately see this reflected in the name of the current program (stored in the `SYS$CURRENT_PROGRAM` variable) and in the output of a **SHOW PROGRAMS** statement.

The program can have parameters if it `is` to be used as the target of a **CALL** statement. These parameters are specified on the **CALL** statement, and their values are copied into the parameter variables when the program is executed by the **CALL** statement. Parameters can optionally have a default value given to them, which means that if the **CALL** statement doesn't provide the parameter, it will take on the default value.

```
100   PROGRAM CALC( VALUE, COUNT=10 )
105   SUM = 0.0
110   FOR I = 1 TO COUNT
120      SUM = SUM + VALUE
130   NEXT I
140   RETURN SUM
```

In this case, if the **CALL** statement that invokes this program does not supply a second parameter for `COUNT`, then the variable will contain 10 by default. If a program does not have parameters in its **PROGRAM** definition, then it can be executed by a **CALL** statement but cannot have parameters passed to it.

The type of the parameters can be optionally specified.

```
100 PROGRAM FOO( INTEGER X, DOUBLE Y )
```

In this example, the parameters have explicit types given to them. When the type is given, whatever data is passed to the program is converted to the given type before being stored in the local argument variable. So the above example guarantees that the first value will be an integer, and the second will be a double, regardless of the values passed to the function. If the type names are not given, then the argument variables take on whatever type the passed parameter was in **CALL** statement that invokes the program.

You can define additional characteristics of the program using the **`DEFINE()`** clause following the program name and parameter specifications. These characteristics affect the way that the program is run or managed by JBasic. The **`DEFINE`** characteristics are:

Keyword	Definition
`SYSTEM_OBJECT`	This program object should be considered owned by JBasic rather than created by the user. This mode is the default for programs loaded from Library.jbasic or from within the JBasic *jar file*.
`STATIC_TYPES`	This program should use static types for variables created while it is running. See the section on **Variables** at the start of this document for more information.
`DYNAMIC_TYPES`	This program should use dynamic types for variables created while it is running. This is the default state.

An example of using the **`DEFINE`** clause might be a program that was originally written for another dialect of BASIC that assumes variable types:

```
100    PROGRAM INVOICE DEFINE(STATIC_TYPES)
110    NAME$ = 1
```

This indicates that variables created by this program have static types. The variable `NAME$` is a string variable when static types are enabled, so the above program segment will result in the value 1 being converted to the string "1" when it is stored, because automatic type conversion occurs when writing values to variables if **`STATIC_TYPES`** is enabled.

PROTECT

The **`PROTECT`** command marks a program as *protected*. A program that is protected cannot be viewed by the end-user as source. This means it cannot be selected as an `OLD` program, or listed with **`LIST`** or **`SHOW PROGRAM`** statements, and the source is not available via the `PROGRAM().LINES[]` array

```
PROTECT name
```

Where "name" is the fully qualified name of the program. A verb or function must include the correct prefix, such as the following example:

```
PROTECT FUNC$ENCODE
```

This marks the user-written function `ENCODE` as protected.

When a protected program is saved to a workspace, the code is not stored as a JBasic program but is stored as compiled instructions. A program that is loaded from a workspace as a protected program remains protected. Once a program is marked protected, it cannot be unprotected, and the source code for the program is lost. *Always make a copy of any program you mark as protected!*

You can protect all programs in the workspace at one time, using the command:

```
PROTECT ALL PROGRAMS
```

After issuing this command, no program currently in the workspace can be listed, disassembled, etc. If you save the workspace, then all source for the programs will be lost.

PUT

The **PUT** command is used to write information to a **BINARY** format file. The information written to the file is stored in variables just like a **PRINT** statement that writes to a file. However, the format of the data stored on the file matches the internal binary representation of the information in the computer, as opposed to a human-readable format. Data written to a **BINARY** file must be identified not only by the variable that contains the information, but the type of data that the information is stored as on the disk file.

For example, a variable `SALARY` might contain the number 5. This can be written to the disk as an integer or as a floating point value. The difference is both in the nature of the data written (is there a fractional part, for example) and also the binary file layout. All data written to a **BINARY** file must be re-read using a **GET** statement using the same binary format.

The binary format is specified in one of two ways: explicitly in the **PUT** statement or via an array of record data types that describes what is to be written.

```
PUT FILE BD, INTEGER ID, STRING(30) N, DOUBLE SALARY
```

In this example, the variables `ID`, `N`, and `SALARY` are written to the file. The variable `ID` is written as an integer, with a possible value in the range of +/- 2^31. The variable `N` is written as a string, with space reserved in the file for 30 characters of data. The variable `SALARY` is written as a double precision floating point value.

Note that while the variable `N` may be any length from zero to 30 characters, space is reserved in the file for 30 characters of data. This will be important when the program needs to position itself in the file and must calculate the location of the binary data. See the documentation on the **SEEK** command for more information. An explicit length is required for **STRING**, **UNICODE STRING**, and **VARYING STRING** specifications. In addition, a **FLOAT** type is permitted which converts the normal JBasic **DOUBLE** into a single-precision value that consumes only four bytes.

In addition to naming variables to be written to the file, you can use expressions. The result of the expression is converted to the given type in the **PUT** statement, as in:

```
PUT FILE BD, INTEGER EMPID + 1000
```

This calculates the value `EMPID+1000` and writes it as an integer to the **BINARY** file. The type of the calculation will be converted as needed.

The record definition could be specified in an array of records as well. Consider the following sample code that constructs the definition of the binary file data and then uses that to **PUT** data to the file:

```
DIM E[3]
E[1] = { TYPE:"INTEGER", NAME:"ID" }
E[2] = { TYPE:"STRING", NAME:"N", SIZE:30 }
E[3] = { TYPE:"DOUBLE", NAME:"SALARY" }

PUT FILE BD, USING E
```

In the example, an array `E` is created with elements for each data item to be written to the binary file. Each element is a record data type, which must contain the fields `TYPE` and `NAME`. The `TYPE` field defines the data type to be written to the file, and the `NAME` field defines the variable name that contains the data. The `SIZE` field is used for `STRING` data types to define how much space to leave in the record for the string value, or to specify the size of the **`INTEGER`** or **`FLOAT`** point value in bytes.

The allowable values for the `TYPE` field are:

TYPE	Description
`"INTEGER"`	An integer value. The default is a 4-byte integer in the range from -2147483648 to 2147483647. An optional size can be given of 1, 2, or 4 to specify byte, word, or integer values.
`"FLOAT"`	A floating point number. The default is a 4-byte single precision value, but a size can be given of 4 or 8 to select single or double precision data.
`"DOUBLE"`	A double-precision floating point number (8 bytes). This is the same as specifying "`FLOAT"` with a `SIZE` value of 8.
`"STRING"`	A string of text, with a specific length allocated in the record as specified by the `SIZE` field
`"UNICODE"`	A string of Unicode (UTF-16) text, with a specific length allocated in the record specified by the `SIZE` field.
`"VARYING"`	A string of text of varying length. The maximum amount of space the string can take is specified in the `SIZE` field. An additional integer value is read or written that contains the actual length of the string in the buffer.
`"BOOLEAN"`	A single byte of data containing 1 or 0 to represent true or false values.

Note that using a record definition array means that the value(s) to be written to the file must reside in variables; they cannot be expressions since the name of the variable must be given in the array.

You can use the **`FIELD`** statement to create a definition of the contents of each data record in the file. This is then automatically used when a **`PUT`** statement writes data. For example,

```
FIELD #1, STRING(30) AS NAME, INTEGER AS ID
PUT #1
```

The **PUT** statement uses the values in the variables **NAME** and **ID** to store data in each record of the file. The **FIELD** stays in effect until another **FIELD** statement creates a new definition or a **CLEAR FIELD** statement removes the field definition completely. See the documentation on the **FIELD** statement for more information.

QUIT

The **`QUIT`** statement terminates JBasic. If you have modified any of the programs in stored memory, JBasic will prompt you to see if you really mean to **`QUIT`** before issuing a **`SAVE`** command.

When JBasic prompts the user if they wish to preserve un-saved changes, it uses the prompt string `SYS$SAVEPROMPT`, which is automatically initialized to the default:

```
There are unsaved programs. You must use the SAVE
command to store them in a workspace.

Are you sure you want to QUIT [y/n]?
```

You can modify the `SYS$SAVEPROMPT` variable to contain any string you wish, and that is used as the prompt string. If you set the variable to an empty string, it disables the prompting entirely, and JBasic will quit without any warning to the user about unsaved programs when a **`QUIT`** command is given.

RANDOMIZE

The **`RANDOMIZE`** statement is used to set the initial “seed” for the random number pseudo variable **`RND`**.

```
RANDOMIZE integer-expression

RANDOMIZE TIMER
```

The first version requires an integer expression. This value is used to provide a starting value for the pseudo-random number function used by the **`RND`** variable. This function guarantees that for any given seed value, the sequence of random numbers will be the same. This allows you to write code that assumes randomness, but test it with a predictable sequence of random numbers.

The second form uses a timer-based internal function to generate the seed value. This results in an unpredictable sequence of numbers from the **`RND`** function, and more closely approximates a true random number.

Note that in all cases, the **`RND`** function generates an artificial random number that is always in the range $-32768 < x < 32767$, and is always an integer value. The **`RANDOM()`** function generates a dramatically more random distribution of values, which also can be coerced into specific ranges using the function parameters.

In place of the **`RND`** variable, you can use the **`RNDVAL()`** function which returns the exact same information.

The **`RANDOMIZE`** statement and the **`RND`** pseudo-variable are provided for compatibility with programs written in GW-Basic or similar dialects supporting these operations.

READ

The **READ** statement is used to access values defined in **DATA** statements and store those values in variables that can be used in the running program.

```
READ X, Y
DATA 3, "Tom"
```

The above statements reads two values and stores them in the variables `X` and `Y`. The values read are the next to items in a **DATA** statement somewhere in the program. The **DATA** statement does not have to be executed, or located near the **READ** statement. See the documentation on the **DATA** statement for more information.

If a **READ** statement is given and there is no **DATA** statement anywhere in the current program, then an error is generated. If a **READ** statement is executed and all the **DATA** statement elements have been read, then the **READ** statement starts again at the first **DATA** statement value found in the program.

The function **EOD()** can be used to determine if the next **READ** statement will read a new value or if an end-of-**DATA** condition will cause it to start at the first **DATA** statement again.

The variable being read can specify *scope qualifiers* to define the attributes of the variable. See the documentation on the **LET** statement for more information on scope qualifiers.

You can specify arbitrarily complex expressions representing the location in which the data value that is read is to be stored. For example, this assumes an array of employee records and sets the value of the employee name in an element of the array:

```
READ EMPLOYEES[EMPID].NAME
```

In this example, the variable `EMPID` is used to locate a specific record in the `EMPLOYEES` array and then set the value of the DATA item read into the member `NAME` in that record. When used in an **READ** statement (or an **LINE INPUT, INPUT**, or **LET** statement), if the array index position or record member does not exist, then it is created.

REM

The **REM** verb is used to indicate lines which are to be treated as comments in some other dialects of BASIC. In JBasic, comments are marked with double-slashes "//" and can be at the end of a line as well as making up the entire line.

```
100 REM This is a test program
110 // It was written by J. DOE
```

Both of these are valid comments in JBasic. However, the first line will have the **REM** keyword removed and replaced with the double-slashes automatically by JBasic. You would see this if you **LIST** the program, or if you **SAVE** the workspace; the **REM** keyword would be replaced by the "//" characters.

Note that you can put comments after a statement as well.

```
...
250 COUNT = COUNT + 1 // Allocate additional inventory
...
```

The statement ignores the "//" and any characters that come after it to the end of the line, so comments can be placed on the line with the statement if you wish, as well as on lines by themselves.

For compatibility with other dialects of BASIC you can also make comments using a single quote character (the apostrophe character). This is converted to the "//" notation automatically. Note that you cannot use this format at the end of a statement, only at the start of a line to mark the entire line as a comment. For example, consider the following source fragment:

```
100 ‘ AVERAGE
110 ‘
120 ‘ Compute the average of a number of grades entered
130 ‘ by the user.
140 DIM GRADES[100]
      ...
```

RENUMBER

Programs created at the console always have line numbers, which identify the sequence in which the statements are stored and executed (see the documentation section regarding *Stored Programs* for more information). The line numbers are arbitrary positive integers. Sometimes, it is convenient to renumber them, particularly to make new room for additional statements to be added between existing statements.

```
RENUMBER [start [increment]]
```

The optional starting line number identifies what the first line number of the program should be. The default is 100. The optional increment indicates what the increment between line numbers should be. The default increment is 10.

*Remember that the **`RENUMBER`** command will change line numbers if they are used as the destination of a **`REWIND, GOTO`** or **`IF`**..**`THEN`** statement.*

RESUME

The **`RESUME`** command is used to restart execution of a program that has stopped due to a **`STOP`** statement or by triggering a breakpoint while running under control of the **`DEBUG`** command.

```
RESUME

   or

RESUME RESET
```

The `RESUME` statement can only be given when a program is under control of the debugger. It causes the program to resume execution after it has stopped. The program will continue until another **`STOP`** statement or breakpoint returns control to the debugger, or the program terminates normally.

Normally, when a breakpoint becomes active (is "triggered") it will stop the program when the trigger occurs. For example, a break point created with **`BREAK WHEN`** `I > 5` will stop execution the first time that the variable I becomes greater than the value `5`. However, subsequent statements will execute even though the value of `I` is still greater than `5`. This is because the breakpoint occurs only when the condition first becomes true.

You can use the **`RESUME RESET`** to reset all breakpoints so they are no longer considered triggered. The next statement will evaluate breakpoints and stop only if a breakpoint becomes triggered on any given statement. This is particularly useful if you have reset the value of `I` and want to cause the debugger to begin checking for the condition of being greater than `5` again.

RETURN

The **RETURN** statement ends execution of a program or function. It may specify a value to be returned to the caller of the function or program. If there is no calling program, then control is returned to the user. For example,

```
RETURN RSLT*2
```

The **RETURN** statement is also used to return from an internal subroutine invoked with a **GOSUB** statement. See the documentation on the **GOSUB** command for examples and more information.

REWIND

The **REWIND** statement is used to direct the next **READ** statement to begin with the first **DATA** statement in the program, even if that is not the next place to **READ**.

```
DATA 101, 102, 103
READ X, Y
REWIND
READ Z
```

In the above sequence, the value of `Z` will be 101, since the **REWIND** tells JBasic to start with the first **DATA** element. Optionally, the program can specify a specific line to mark as the next place to read by using a line number or statement label, as in the following example:

```
100   DATA 101, 102, 103, 104
110   DATA 201, 202, 203, 204
120   READ X, Y
130   REWIND 110
140   READ Z
```

The first two **READ** statements will read the first two data items (`101` and `102`). The **REWIND** statement then indicates that the next **READ** statement should begin with the first **DATA** item at line `110`. Therefore, the next **READ** statement will store the value `201` in the variable `Z`.

See the documentation on the **READ** and **DATA** statements for more information.

You can also use the **REWIND** statement to rewind a **BINARY** format file to the first byte of the file, so that the next **GET** or **PUT** statement will read or write starting at the beginning of the file.

```
100   REWIND FILE FOO
110   GET FILE FOO, USING BOB
```

The **REWIND** statement repositions the file to the start of the file so the **GET** statement reads the record located at the first byte of the file. This is equivalent to using the **SEEK** statement to position at byte zero.

You can also **REWIND FILE** on an `INPUT` mode file, and it causes the next **INPUT** or **LINE INPUT** operation to read data from the first line of the input file. This has no effect on files that are connected to a `TERMINAL` device such as the console.

It is an error to use the **REWIND** operation on a file that is not opened for **BINARY** or **INPUT** modes.

RUN

The **`RUN`** statement executes a stored **`PROGRAM`**. By default, the current program is run. The current program is the last program that was **`RUN`**, or the program identified by the last **`NEW`** or **`OLD`** statement. Optionally you can specify the name of an existing stored program to **`RUN`**.

```
RUN

RUN BENCHMARK
```

When a program is executed via a **`RUN`** command, the variable `$MODE` in the local symbol table is set to "RUN". This helps differentiate between programs that were executed via the **`RUN`** command versus executed via a **`CALL`** statement

You can add the **`DEBUG`** keyword after **`RUN`** to indicate that the program is to be run under control of the debugger. See the introductory section on debugging for more information.

```
RUN DEBUG PI_TEST
```

This runs the program `PI_TEST` under control of the debugger. If the name of the program is omitted, then the current program is run with the debugger.

SAVE

The **`SAVE`** command saves the current program to a disk file in a human-readable text format, suitable for use later with a **`LOAD`** command to read a program into memory.

```
SAVE "file name"
```

The filename must be explicitly specified; there is no default name. You can save a program into a different file than the one it was loaded from, for example.

Note that the **`SAVE`** command saves only the current program to disk. If you wish to save all programs at once, you can use the **`SAVE WORKSPACE`** to save all programs into a single workspace file.

SAVE PROTECTED

You can save just the current program as an encoded XML file that can be loaded via the **`LOAD`** command or transmitted to another program as an XML file. The contents of the file are not human-readable but conform to the XML standards.

```
SAVE PROTECTED "file.xml"
```

This saves the current program as "`file.xml`". The source code for the program is not stored with the program, only the information necessary to reconstruct the internal byte code representation of the program. Unlike the PROTECT command, this command does not modify the program in memory; the current program still exists in its unprotected form - only the disk representation created is protected.

The resulting file can be read back into memory using the **`LOAD`** command which automatically recognizes when a file contains an XML protected program definition as opposed to a text program description. The resulting program in memory *will* be protected at that point.

SAVE WORKSPACE

The **`SAVE WORKSPACE`** command saves the current workspace; that is, all programs currently in stored program memory that have been modified by the user. This includes programs previously brought into memory with the **`LOAD`** command as well as any programs created or changed in this session. The workspace contains all such programs.

```
SAVE WORKSPACE [ "workspace name" ]
```

If a name is not given, then the default is "Workspace.jbasic". If a name is given, it becomes the new default name for this session; subsequent **`SAVE WORKSPACE`** commands will be written to the same named workspace file.

When JBasic first starts up, it automatically loads all programs it finds in the default workspace file `“Workspace.jbasic”` located in the current user's home directory, if that file exists. You can determine the home directory used via **`PROPERTY("user.home")`** which corresponds to the Java property of the same name.

You can determine the current default workspace name by looking in the `SYS$WORKSPACE` global variable which contains the full path name of the workspace.

SAVE XML

You can save just the current program as an XML file that can be loaded via the **`LOAD`** command or transmitted to another program as an XML file.

```
    SAVE XML "file.xml"
```

This saves the current program as "`file.xml`". This command cannot be placed in a running program; it can only be executed from the console or via the JBasic run() method in an embedded program.

The resulting file can be read back into memory using the **`LOAD`** command which automatically recognizes when a file contains an XML program definition as opposed to a text program description.

SEEK

The **`SEEK`** command is used to position a **`BINARY`** file to a specific location in preparation for a **`GET`** or **`PUT`** operation to read or write binary data. The **`BINARY`** file has a *file pointer* which is an integer value describing which byte to read or write next. The first byte in the file is numbered zero.

A **`BINARY`** file is organized as a stream of bytes. A byte is the smallest addressable unit of memory in a computer, and typically holds a value from 0-255. This can hold a value like true or false, or other small amounts of data. Multiple bytes are used together to hold larger or more complex values. For example, an integer value takes four bytes, which allows it to store numbers in the range of -2147483648 to 2147483647. Similarly, character data is stored using one byte for each character, and optionally includes four additional bytes to hold the length of the string. Here is the complete table of types and how many bytes it takes to store each one:

TYPE	BYTES	Description
`"INTEGER"`	`1, 2, or 4`	An integer value. The default is a 4-byte value in the range from -2147483648 to 2147483647. An optional `SIZE` value may specify sizes of 1, 2, or 4 to indicate byte, word, or integer values.
`"DOUBLE"`	`8`	A double-precision floating point number. This is the same as `FLOAT` with a `SIZE` value of 8.
`"FLOAT"`	`4 or 8`	A floating point number. The default is a 4-byte single precision number. However, an optional `SIZE` value may specify 4 or 8, indicating single or double precision.
`"STRING"`	`length`	A string of text. One byte is used for each character. The size is defined by the `SIZE` record field.
`"UNICODE"`	`length * 2`	A string of UNICODE text. Each character takes two bytes of data.
`"VARYING"`	`length + 4`	A varying length string of text. One byte is used for each character, plus an additional 4 bytes to hold the actual string length within the fixed-sized buffer.
`"BOOLEAN"`	`1`	A single byte of data containing 1 or 0 to represent true or false values.

An important reason why a program might use a **`BINARY`** file rather than a standard text file is that the storage for a value takes a known number of bytes, and that known number of bytes means that you can calculate the position of any value in the file.

Imagine a file that has the following data, representing employee information:

Value	Description
"NAME"	A 30-character name for an employee
"ID"	An integer with the employee id number
"WAGE"	A double precision value with the hourly wage.

To express this, let's create an array definition that describes the record, as documented in the sections on the **GET** and **PUT** statements:

```
DIM EMPREC[3]
EMPREC[1] = { NAME:"NAME", TYPE:"STRING", SIZE:30 }
EMPREC[2] = { NAME:"ID", TYPE:"INTEGER", SIZE:4 }
EMPREC[3] = { NAME:"WAGE", TYPE:"FLOAT", SIZE:8 }
```

Because we know the size of each type, we can calculate the width of the record in bytes. That is, we can calculate how many bytes it take to store the entire record. This would be

```
RECSIZE = 30 + 4 + 8
```

Given this calculation, it is now possible to position the **BINARY** file to write any given employee's information. If the first byte in the file is numbered zero, then the calculation `RECSIZE*(RECNUM-1)` will calculate the byte position for record numbers starting at record one. For example,

```
PROGRAM WRITE_EMP( BD, NAME, ID, WAGE )
DIM EMPREC[3]
EMPREC[1] = { NAME:"NAME", TYPE:"STRNG", SIZE:30 }
EMPREC[2] = { NAME:"ID", TYPE:"INTEGER", SIZE:4 }
EMPREC[3] = { NAME:"WAGE", TYPE:"FLOAT", SIZE:8 }
RECSIZE = 30 + 4 + 8

SEEK FILE BD, RECSIZE*( ID-1 )
PUT FILE BD, USING EMPREC
RETURN
```

In this example, a program is written that will write an employee record to the file. The employee identification number is also the record number in the file. The **SEEK** statement positions the file to the location corresponding to the employee number, and then the **PUT** statement writes the data to the file at the current position just set by the **SEEK** statement. A program can also use the **FILEPOS**`(ID)` function to get the position of the file indicated by the file identifier variable `ID`.

You can also use a **FIELD** statement to create a record definition, and then seek to the specific record in the file using that record definition. For example,

```
FIELD EMPREC AS STRING( 30 ) NAME, INTEGER(4) ID
SEEK FILE BD, USING EMPREC, N
```

In this example, the file is positioned to record N, which is a 1-based position indicator. That is, the value of N should be 1 for the first record in the file, 2 for the second record, etc. The **FIELD** definition is used to automatically calculate how large each record is (using the same formulas described above) and the file is positioned accordingly.

You can also use the **FIELD** statement to create a definition of the contents of each data record in the file. This is then automatically used when a **SEEK** statement calculates a file position. For example,

```
  FIELD #1, STRING(30) AS NAME, INTEGER AS ID
  SEEK #1, N
```

The statement calculates the size of the **FIELD** definition because it is bound to the file specification, and computes the correct position for record N using this size information. The **FIELD** stays in effect until another **FIELD** statement creates a new definition or a **CLEAR FIELD** statement removes the field definition completely. See the documentation on the **FIELD** statement for more information.

SIGNAL

The **`SIGNAL`** command generates a runtime error. This can be used in a program to simulate an error, or to invoke an **`ON`** unit previously defined with the `ON` statement.

```
SIGNAL code [( argument )]
```

The "code" is the name of a predefined status code like `SYNTAX`, or a user-defined message created with the **`MESSAGE`** statement. If the message has a substitution value used in formatting a message, you can specify that after the signal code.

For example:

```
  SIGNAL SYNTAX("unexpected comma")

  SIGNAL ARRAY

  SIGNAL USING( BOB ) ("unexpected data")
```

The last example uses the expression (in this case, a variable `BOB`) to define the code. If `BOB` was equal to "`SYNTAX`", then the first and third examples have similar output.

SLEEP

The **SLEEP** command causes the current thread to pause execution for a specified amount of time. This is most often used when a thread needs to wait a short period of time to allow other threads to complete execution.

```
SLEEP count  [unit]
```

The *unit* parameter is used to describe the units that *count* represents. The default is **SECONDS** if no unit is given; keywords can be singular or plural as needed for readability. The effect of the unit keyword is to multiply the count by a suitable value to convert it to seconds (or fractions of a second). The allowed keywords and multipliers are:

Keyword	Multiplier
MILLISECONDS	0.001
SECONDS	1.0
MINUTES	60.0
HOURS	3600.0
DAYS	86400.0
WEEKS	604800.0

The "*count*" parameter is the number of *units* to sleep. The *count* can be an integer value or a double value; sleeping for fractions of a unit such as a second is permitted. The actual minimum amount of time that the thread will sleep depends on the implementation of Java and the underlying operating system but is usually accurate to within a millisecond (0.001 seconds).

SORT

The **SORT** statement is used to sort an array into ascending order. The array elements must all be of the same type - you cannot mix strings and numbers. Additionally, the array may only contain scalar values (strings and numbers) and may not contain records or other arrays.

```
X[1] = "Tom"
X[2] = "Debbie"
X[3] = "Robert"
X[4] = "Nancy"
SORT X
PRINT X
```

The above will result in a printout similar to

```
[ "Debbie", "Nancy", "Robert", "Tom" ]
```

You can also use the **SORT()** function with an array name as the parameter, and it will return a new array in sorted order. Using the above array example,

```
Y = SORT(X)
```

This would result in a new array called `Y` that is a sorted version of `X`.

You can also use the **SORT** statement to sort an array of records, and specify the field in the record(s) to be used as the sort key value.

```
X[1] = { NAME: "Bob", AGE: 45 }
X[2] = { NAME: "Sue", AGE: 35 }
X[3] = { NAME: "Dave", AGE: 36 }
X[4] = { NAME: "Amy", AGE: 42 }
SORT X BY NAME
```

The **BY** clause identifies a field name in the records to be sorted. The example above will result in the record with `NAME:"Amy"` being the first element in the array, and `NAME:"Sue"` being the last. The field values must all be the same type (just like sorting a standard array) and all the records must have a field of the given name.

STEP

The **STEP** command is used to control program execution in the debugger. See the introductory section on debugging to get more information on how to run a program under control of the debugger. The **STEP** command can only be used when the debugger is active.

The **STEP** command has several formats, each of which are described here.

```
STEP [count]
```

This steps (executes) one or more statements in the program. The value *n* defines how many statements to execute. If not specified, then one statement is executed. This is the same as just pressing return at the DBG> prompt without entering a command.

```
STEP INTO
```

By default, when a program calls another program (as a **CALL** statement, or a function call, or a verb) the debugger runs the called program in its entirety as a single **STEP** operation. However, **STEP INTO** can be entered and if the next statement to execute invokes another program, that program is executed under control of the debugger as well. This command only takes effect on the statement about to be executed.

```
STEP RETURN
```

If a **STEP INTO** has been issued and the user is debugging a program that was called by another program, it may be desirable to resume execution at the caller's program again. The **STEP RETURN** command causes the program to run until the next `RETURN` statement, where the debugger regains control.

STRING

The **`STRING`** statement declares one or more variables as string variables, and assigns them initial values. Optionally, the value can be declared as an array of string values by using an array size. For example,

```
    STRING NAMES[10], STATE=”NC”
```

This statement creates an array with ten elements, each of type `STRING`. The array is initialized to the default value for the type, so in this case it is an array of empty string values. The array size must be specified using square brackets, not parenthesis.

The statement also creates a single `STRING` value named `STATE`, which is initialized to a specific value of `“NC”`.

This statement has the same effect of using the **`DIM`** statement with an explicit type of **`STRING`**, but also allows you to specify an initial value for each item.

SUB

The **SUB** statement defines a local subroutine or function in the current program. Unlike a **PROGRAM** or **FUNCTION** statement that causes a new program object to be created, a **SUB** routine is part of the current program, and can only be called or invoked from the current program.

```
PROGRAM FOO
X = DBL(3.5)
PRINT “Result is “; X
SUB DBL( V )
Y = V * 2
RETURN Y
```

In this program named `FOO`, there is a local subroutine named `DBL`. This can be used either in a **CALL** statement as a subroutine, or as a function. When used as a function, it *must* return a value or an error occurs. In the example above, the function `DBL` returns its argument multiplied by two. You could implement this with a statement function (See the **DEFFN** statement) but a SUB routine can be an arbitrarily complex muti-statement function.

The code above could be executed with a **CALL** statement rather than as a function. In this case, the $MODE local variable would contain the value “`FUNCTION`” rather than “`CALL`”, allowing the **SUB** routine to determine how to return control to the calling program if needed.

When a **SUB** routine is called or invoked as a function, the variable `$THIS` is set to the name of the called routine, and `$FROM` is set to the name of the main program. In both cases, the global variable `SYS$CURRENT_PROGRAM` will refer to the containing program `FOO`.

Note that in the example above, execution of the **PRINT** statement is followed by terminating the program. You cannot “execute” a **SUB** statement; there is an implied **END** statement before the **SUB** statement that prevents flow of control into the subroutine other than by a **CALL** or function invocation.

Like a **PROGRAM** or **FUNCTION**, the **SUB** routine can have multiple arguments, a varying argument list, and explicit types on the arguments. See the documentation on the **PROGRAM** statement for more information.

```
100   PROGRAM FOO2
110   X = CALC(12)
120   Y = CALC(15, 5)
150   PRINT X, Y
200   SUB CALC( VALUE, COUNT=10 )
205   SUM = 0.0
210   FOR I = 1 TO COUNT
220      SUM = SUM + VALUE
230   NEXT I
240   RETURN SUM
```

In this case, if the function invocation that calls the **SUB** routine CALC does not supply a second parameter for COUNT, then the variable will contain 10 by default. If a program does not have parameters in its **SUB** definition, then it can be executed by a **CALL** statement but cannot have parameters passed to it.

The type of the parameters can be optionally specified.

```
200 SUB CALC( INTEGER X, DOUBLE Y )
```

In this example, the parameters have explicit types given to them. When the type is given, whatever data is passed to the **SUB** routine is converted to the given type before being stored in the local argument variable. So the above example guarantees that the first value will be an integer, and the second will be a double, regardless of the values passed to the function. If the type names are not given, then the argument variables take on whatever type the passed parameter was in **CALL** statement that invokes the routine.

SUBTRACT

The **SUBTRACT** statement subtracts the value of an expression from an existing variable.

```
      SUBTRACT (HOURS*RATE) FROM PAY
```

The above statement calculates the expression and subtracts it to the existing value of `PAY`. If the variable `PAY` does not exist, then this statement generates a runtime error. This is essentially equivalent to the statement **LET** `PAY = PAY – (HOURS*RATE)`.

Note that if this expression uses pre- or post-increment or decrement operators in the target expression, they are processed after the statement completes the storage of the new value in the target. For example,

```
    B = 3
    SUBTRACT X+B FROM EMPS[B++]
```

The increment operation of `B` will take place after the difference of the element less `X+B` is stored in the array `EMPS` at the location of the current value of `B` (3 in this case).

The **SUBTRACT** statement can also be used to delete a substring from a string value:

```
    X = "FANCY RED APPLES"
    SUBTRACT "RED " FROM X
```

The result is that the variable contains the string "FANCY APPLES". This requires that both subtrahends be string values for the difference to be a new string value. Only the first instance of the substring is deleted from the string.

SYSTEM

The **SYSTEM** statement executes a single statement in the native command line environment. For Windows, this is a DOS emulation command. For Unix systems, it is a command in your current default shell. For VMS, it is a DCL command.

```
SYSTEM expression
```

The command to be executed is defined by the expression, which can be any string expression. It can be a simple quoted string or a more complex string expression. The system variable `SYS$STATUS` is created if it does not exist and is set to a status record describing how the subprocess completed. The `CODE` field is always "`*SYSTEM`" and the `PARM` field is a string representation of the numeric return code from the executed command. Normally zero means success for all systems except OpenVMS where any odd number means success.

Here is an example of using this statement on a Unix computer:

```
BASIC>  system "ls -l"

total 960
-rw-r--r--    1 tom  tom     741 Aug  9 14:53 GET-STARTED.TXT
-rw-r--r--    1 tom  tom  138459 Oct  2 08:49 JBASIC-HELP.TXT
drwxr-xr-x    4 tom  tom     136 Apr  5 10:52 META-INF
-rw-r--r--    1 tom  tom    2201 Jun  2 08:59 build.xml
drwxr-xr-x    4 tom  tom     136 Aug  4 15:55 src
```

It should be noted that the command line will be broken into a command and arguments by the underlying shell code, using spaces as a delimiter. So the example above is really sending the command "`ls`" and the argument "`-l`" to the Unix command shell. If the arguments might contain spaces, then instead of a single string value that is the entire command, the **SYSTEM** command can be used with an array of strings.

```
SYSTEM [ "ls", "-ls", FN ]
```

In this example, the command is "`ls`", the first argument contains the switches "`-ls`", and the second argument contains the filename variable. Because it is expressed as an array, the filename isn't processed in any way and is passed as-is to the command, even if it contains spaces or other special characters such as Unicode characters.

This form of the **SYSTEM** command requires either an array expression or a comma-separated list of string values.

TABLE

The **TABLE** statement is used to define a TABLE data type. For an overview of the use of the TABLE data type, see the section on **Tables** in the introductory section of this manual. A TABLE is an array of records where the member names and types of each row are identical. That is, each row is guaranteed to have the same member names and data types as every other row. This is used to store rectangular data in a fashion similar to a conventional database.

A TABLE must be defined before it can be used because you must indicate the column names and data types.

```
    TABLE EMP_DATA AS INTEGER ID, STRING LAST, STRING FIRST, DOUBLE RATE
```

This example creates a table named `EMP_DATA` which has four columns (`ID`, `LAST`, `FIRST`, and `RATE`) with the given JBasic data types.

Once this table is created you can add rows to it using the "+" operator to add an array or record type (as long as the type, number, and - in the case of a record - member names match). You can also use the **INPUT ROW OF** notation to read a row in from a file or the console.

If you **PRINT** a table, the output is formatted with the column headings created using the column names from the **TABLE** declaration.

UNLOCK

The **UNLOCK** command is used to release locks, which are objects used to protect critical regions of a JBasic program. Specify one or more lock names in a list to be unlocked.

```
UNLOCK COUNT_L, COUNT_Q

       or

UNLOCK ALL LOCKS
```

The first statement release two locks `COUNT_L` and `COUNT_Q`. The locks must exist and be held by the current thread or an error is reported. The second example releases all locks held by the current thread unconditionally.

When a program has created a dynamic lock name, where the name is stored in a string variable, the **UNLOCK** statement will accept a **USING** clause, as in:

```
UNLOCK USING( “LOCK_” + LCKID )
```

This example assumes that an integer value `LCKID` will be appended to the string “`LOCK_`” to indicate the specific lock being released.

Note that you cannot **UNLOCK** a lock that you didn't lock. If you **LOCK** the same named lock more than once, you must unlock it the same number of times before it is available for another process to use. If a thread exits while it still holds a **LOCK** then that lock is released.

See the documentation for the **LOCK** statement for examples of how to use locks in a JBasic program running with threads. The **SHOW LOCKS** command can be used to display the locks currently in the system. The **CLEAR LOCK** command is used to delete locks, and the **LOCKS()** function is used to get an array of records describing all known locks.

UNTIL

The **UNTIL** statement defines the end of a **DO...UNTIL** block, and contains the expression that would end the loop. See the documentation on the **DO** statement for more information and examples like the following:

```
100 DO
110   SUM = SUM + VALUE[I]
120   I = I + 1
130 UNTIL SUM > 100.0
```

Looping statements like **DO...UNTIL** cannot be executed directly from the command line but can only appear in programs.

For compatibility with other dialects of BASIC you can use the keyword **LOOP UNTIL** as a synonym for **UNTIL**, as in the following example:

```
100 DO
110   SUM = SUM + VALUE[I]
120   I = I + 1
130 LOOP UNTIL SUM > 100.0
```

VERB

The **VERB** statement appears in a text file and identifies a block of statements to be stored as code to be executed when the corresponding command (a "verb") is given in JBasic. This is very similar to a **PROGRAM** except it will be automatically run by JBasic when the verb is given as a statement.

```
VERB SUPPLY
```

This defines a verb that will be named `SUPPLY`. This must be the first line of a new program that you create. In fact, the **NEW** command will automatically create a first line containing a **PROGRAM** statement for you. You will need to modify this statement to be a **VERB** statement if you are writing a statement extension to JBasic.

A **VERB** has access to the rest of the statement text that follows it. The rest of the line is parsed into individual tokens, and these are stored in the `$ARGS[]` array that is passed to the program. So if you entered the statement

```
SUPPLY DEPOT 3
```

The element `$ARGS[1]` would contain "DEPOT" and `$ARGS[2]` would contain "3". Note that the arguments are always strings, but you can use them as numbers if you know they contain only numeric values. So `$ARGS[2] * 2` would be equal to 6 in the above example.

```
SUPPLY A <> 3
```

If your statement has punctuation, the punctuation will be parsed as individual tokens as well. So the above example will have three tokens in the array: `{"A", "<>", "3"}`

In addition to the array of individually parsed tokens, the entire text of the command as given by the user is available in the string variable `$COMMAND_LINE`.

You can define additional characteristics of the verb you create by using the **DEFINE()** clause following the function argument list. See the documentation on the **PROGRAM** statement for more details.

WHILE

The **`WHILE`** statement defines the end of a **`DO...WHILE`** block, and contains the expression that would end the loop. See the documentation on the **`DO`** statement for more information and a description of how examples like this work:

```
100 DO
110   SUM = SUM + VALUE[I]
120   I = I + 1
130 WHILE I < MAXVALUE
```

Looping statements like **`DO...WHILE`** cannot be executed directly from the command line but can only appear in programs.

For compatibility with other dialects of BASIC you can use the keyword **`LOOP WHILE`** as a synonym for **`WHILE`**, as in the following example:

```
100 DO
110   SUM = SUM + VALUE[I]
120   I = I + 1
130 LOOP WHILE I < MAXVALUE
```

Built-In Functions

This section contains a short description of each of the built-in functions that are available in the JBasic language to use as part of any expression. A function is identified by a function name, followed by zero or more arguments in parenthesis. If more than one argument is present, the arguments must be separated by commas.

There are numerous dialects of BASIC, and each have slightly different lists of available functions. More confusingly, they also can have different spellings for the same function. The spellings documented here are the preferred spellings for JBasic programs. However, often an alternative spelling is available to match other dialects. For example, the **LENGTH**() function can be written a **LEN**(), and the **LEFT**() function can be written as **LEFT**$(). The global variable `SYS$FUNCTION_MAP` contains an array of records defining "old" and "new" names for functions, and is used to map function names to the preferred format. You can add to this array if you need to specify other function name mappings.

The remainder of this section describes each function.

ARRAYTOSTR()

```
sval = ARRAYTOSTR( array )
```

Returns an string, where each element of the array is has been concatenated into the string. The string length will be the sum of the length of each array element.

ABS()

```
val = ABS( val )
```

Returns the absolute (positive) value of the argument, which must be an integer or floating point number.

ARCCOS()

```
fval = ARCCOS( val )
```

Returns the inverse cosine of the value *val* expressed in radians.

ARCSIN()

```
fval = ARCSIN( val )
```

Returns the inverse sine of the value *val* expressed in radians.

ARCTAN()

```
fval = ARCTAN( val )
```

Returns the inverse tangent of the value *val* expressed in radians.

ARRAY()

```
aval = ARRAY( v1 [, v2…] )
```

Returns an array whose members are the arguments. The first argument is the first element of the array, the second argument is the second element of the array, etc.

ASCII()

```
ival = ASCII( string )
```

Returns an integer containing the ASCII or UNICODE value of the first character of the given string. If the string is empty, the result is zero.

BASENAME()

```
sval = BASENAME( file-name-string )
```

Returns a string containing the base file name of a file name string. For example, returns “myprogram” for the file name string “/Users/tom/myprogram.jbasic”.

BINARY()

```
sval = BINARY( integer )
ival = BINARY( string )
```

Returns a string containing the integer value converted to hexadecimal notation. This function is the same as calling RADIX() with a radix value of 16. In the second case, the string is processed as a binary value and the result is returned as an integer. For example, `BINARY("101")` results in 5.

BOOLEAN()

```
bval = BOOLEAN( val )
```

The argument is converted to a Boolean (true/false) value. If the value is a number, then zero is the same as false, and non-zero means true. If the value is a string, then the value "true" is converted to true, and all other strings are false.

BYNAME()

```
rval = BYNAME( input-string, name-array )
```

The first argument is a string containing one or more named values, separated by commas. This is the same input format as the **`INPUT BY NAME`** clause. The second argument is an array of unique string names. These are the names of the value expected to be found in the input buffer. The result is a record containing members for each expected variable. In addition, a member named `_UNEXPECTED` contains all other input values found in the input string that were not enumerated in the name array. If the input buffer contains values that are invalid syntax, the global variable `SYS$STATUS` is set with the error message triggered during processing of the input buffer.

CEILING()

```
ival = CEILING( val )
```

If *`val`* is an exact integer, the integer is returned. If *`val`* is a floating point value, then it is rounded up to the next largest integer and that integer is returned.

CHARACTER()

```
sval = CHARACTER( integer-expression )
```

Returns a string containing a single character, which is the character encoded value of the integer-expression. For example, the integer value 65 translates in UNICODE to the single character string "A".

CIPHER()

```
sval = CIPHER( string-expression [, key-string] )
```

Returns a string containing cipher (encrypted version) of the string expression. If a key string is given, then that key string is used to construct the cipher, and must be given again when deciphering the string. If a key-string is not given, then an internal key string is used. This function requires that installation of the java.crypt cryptography package on your system or an error is generated when using this function.

COS()

```
fval = COS( val )
```

Returns the cosine of the value *val* expressed in radians.

CSV()

```
sval = CSV( argument-list )
```

Returns the arguments expressed as a comma-separated list. If an argument is an array, each element of the array is returned as a comma-separated value. If an argument is a record, the record member values are returned in alphabetical order by member name. If not specified, the delimiter defaults to a comma, but can be explicitly set to some other string by setting the global variable `SYS$CSVDELIMITER` to the desired string value.

DATE()

```
val = DATE([date-value [,format-string]])
```

By default, `DATE()` returns a string containing the formatted value of the current date and time. The first optional parameter is the date value, and the second optional parameter is a format specification.

If the date is given as a floating point number, it must be the number of milliseconds since the start of the epoch, and that date is formatted as the date value. If a string is given as the first argument, then the string is parsed to extract a valid date value from the string and returned as a double that contains the date value encoded in milliseconds.

The second argument is the format pattern used to format or parse the data. This is a string containing one or more date elements. Values not in this list are processed literally; i.e. a "/" character is just copied to the output string when the format is "`MM/dd/yyyy`", for example.

Code	Description	Example Output
`G`	Era designator	`AD`
`y`	Year	`1996; 96`
`M`	Month in year	`July; Jul; 07`
`w`	Week in year	`27`
`W`	Week in month	`2`
`D`	Day in year	`189`
`d`	Day in month	`10`
`F`	Day of week in month	`2`
`E`	Day in week	`Tuesday; Tue`
`a`	Am/pm marker	`PM`
`H`	Hour in day (0-23)	`0`
`k`	Hour in day (1-24)	`24`
`K`	Hour in am/pm (0-11)	`0`
`h`	Hour in am/pm (1-12)	`12`
`m`	Minute in hour	`30`
`s`	Second in minute	`55`
`S`	Millisecond	`978`
`z`	Time zone	`PST; GMT-08:00`
`Z`	Time zone	`-800`

The default format string is "`EEE, d MMM yyyy HH:mm:ss Z`" if the second argument with the formatting pattern is not given. This which results in output like "`Sun, 28 Sep 2008 20:05:34 -0400`".

DECIPHER()

```
sval = DECIPHER( string-expression [, key-string] )
```

Returns the argument deciphered to its original form. If a key string was given when the string was originally encoded, this same key string must be given again in the **`DECIPHER()`** invocation. If the wrong key string is given, the result is always an empty string rather than the decrypted value.

DOUBLE()

```
fval = DOUBLE( val )
```

Returns the argument converted to a double precision floating point number.

EOD()

```
bval = EOD()
```

Returns a Boolean value indicating if there is more **`DATA`** that can be used to satisfy a **`READ`** statement. If the value is true, then it means that end-of-data has been reached, and the next **`READ`** will access the first **`DATA`** item again. If the value is false, then there is still at least one more **`DATA`** item to read before starting over. See the documentation on **`DATA`** and **`READ`** for more information.

EOF()

```
bval = EOF( file-identifier )
```

Returns a Boolean expression indicating if the named file (as given by the file identifier from the **`OPEN`** statement) is positioned at the end-of-file or not. If the function returns false, then it is possible to read more data from the file without error.

EXISTS()

```
bval = EXISTS( file-name-string )
```

Returns a Boolean value to indicate if a file exists or not, as defined by a string expression containing its name.

EXP()

```
val = EXP( value )
```

Returns the value of "e" raised to the given exponent.

EXPRESSION()

```
val = EXPRESSION( string )
```

Returns the results of evaluating the expression in the string. For example, **`EXPRESSION`**("3+5") returns the value 8. The expression can be a string or numeric expression, and reference any active variable. The result type is based on the expression type.

EXTENSION()

```
sval = EXTENSION( file-name-string )
```

Returns a string containing the file name extension of a file name string. For example, returns ".jbasic" for the file name string "/Users/tom/myprogram.jbasic".

FILEPARSE()

```
rval = FILEPARSE( file-name-string )
```

Returns a record containing the individual elements of the filename as string values in the record. This functions independently of the underlying native file system; that is, this function works regardless of whether the call is made on Mac, Windows, or Unix systems. The resulting record has the following fields, and uses the example of a Unix file name of "/Users/tom/myprogram.jbasic":

Name	Description
`EXTENSION`	The file extension, such as ".jbasic"
`NAME`	The file name without the extension, such as "myprogram"
`PATH`	The path that contains the file, such as "/Users/tom"

FILES()

```
array = FILES( path-string )
```

Returns an array with the name of each file located in the path given by the path string. If the directory is invalid or empty, then an empty array is returned. This function is different from **`FILETYPES()`** in that it just returns an array of string names, where **`FILETYPES()`** returns a full descriptive record for each file in the path.

FILETYPE()

```
array = FILETYPE( filename )
```

Returns a record describing the file indicated by the string parameter. The array will be empty if the file is invalid.

The record has the following fields:

Name	Description
`PATH`	The directory path of the file.
`NAME`	The name of the file itself.
`READ`	A Boolean indicating if the file can be read.
`WRITE`	A Boolean indicating if the file can be written.
`HIDDEN`	A Boolean indicating if the file is normally hidden.
`FILE`	A Boolean indicating if the object is a file.
`DIRECTORY`	A Boolean indicating if the object is a directory.

An example record might look like `{ PATH: "/Users/tom", NAME: "jbasic", READ: true, WRITE: false, DIRECTORY:true, FILE:false }.` This indicates a that "jbasic" is a directory, can be read but not written, and is not hidden.

Note that **`FILETYPES`**`(path)` returns an array of the same kind of information that **`FILETYPE`**`(file)` returns. **`FILETYPE`**`()` returns the information about a single file; **`FILETYPES`**`()` returns the information about all files in a directory.

FILETYPES()

```
array = FILETYPES( path )
```

Returns an array describing the files found in the file system path described by the string parameter. The array will be empty if the path is invalid or if the directory has no files.

The resulting array has a record in each element, of the same kind of information as the **`FILETYPE()`** function.

Note that **`FILETYPES`**`(`*`path`*`)` returns an array of the same kind of information that **`FILETYPE`**`(`*`file`*`)` returns. **`FILETYPE()`** returns the information about a single file; **`FILETYPES()`** returns the information about all files in a directory.

FLOOR()

```
dvalue = FLOOR( dvalue )
```

Returns the arithmetic floor of the argument; the next smallest cardinal integer value less than the argument if the argument is not already an integral value.

FORMAT()

```
sval = FORMAT( expression, format-string )
```

The result of the expression is formatted using the format string. The format string indicates how the data (which must be numeric) will appear by using sequences of special characters to define how each character of the output will appear.

The format characters are described in the following table.

Character	Description
#	Represents a digit. If left of a decimal point, this will be blank if there is no significant digit to its left. If right of the decimal point, this will be zero of there is no corresponding significant digit to the right.
0	Represents a digit. This will always be zero if there is no significant digit in this place in the string.
.	Indicates where the decimal point is to appear in the formatted value string.

Character	Description
$	Indicates a dollar-sign character that is to appear at this position. If there are more than one $ in the string, then the dollar sign "floats" and will appear directly to the left of the first non-zero digit.
( )	When parenthesis are included in the format, they enclose the digit format. If the formatted value is positive, then blanks are displayed in these positions. When the value is negative, then the parenthesis are printed, allowing accounting-style designation of the sign of the number.

The following table contains some examples of format operations on various values.

Format Operation	Resulting String
`format( 3.5, "###.##")`	`3.50`
`format( 3.5, "000.##")`	`3.50`
`format( 3.5, "$$$.##")`	`$3.50`
`format( -3.5, "(##.##")`	`( 3.50)`
`format( 6, "0000" )`	`6`
`format( 22, "###")`	`22`

The result of the format function call is a string that can be used to print output to the console or a file, or for other display purposes. You can use the same format operations in a **`PRINT USING`** statement, documented elsewhere in this guide, which lets you format multiple values at one time using a single format string, which can also include descriptive text or labels.

GETPOS()

```
ival = GETPOS( fileid )
```

Returns an integer with the current file position of the **`BINARY`** format file identified by the file identifier.

HEXADECIMAL()

```
sval = HEXADECIMAL( integer )
ival = HEXADECIMAL( string )
```

The first case returns a string containing the integer value converted to hexadecimal notation. This function is the same as calling RADIX() with a radix value of 16. In the second case, the string value is processed as a hexadecimal value and the result is the decimal equivalent as an integer value.

INTEGER()

```
ival = INTEGER( numeric-expression )
```

Returns the numeric expression converted to an integer data type. If the value is already an integer then no work is performed. Otherwise, the value is converted according to the standard JBasic rules for conversion.

INPUT()

```
aval = INPUT( format-spec, input-buffer )
```

Uses an input specification to process a string buffer, and returns an array of values that were processed from the input buffer. The type of the array elements is determined by the input specification elements. See the documentation on the **`INFORMAT`** statement for more information.

ISOBJECT()

```
bval = ISOBJECT( variable )
```

Returns a Boolean flag indicating if the given variable is an **`OBJECT`** created by the **`NEW()`** function or by a **`CLASS`** definition.

LEFT()

```
sval = LEFT( string, count )
aval = LEFT( array, count )
```

Returns a string containing the 'count' left-most characters of the string. If the count is less than 1 then an empty string is returned.

In the second form, the operation can be performed on an array. In this case, the result is an array that contains the first *count* elements of the first parameter, which must be an array. For example, the expression **`LEFT`**`([“T”, 3, 8.2], 2)` results in the array `[“T”, 3].`

LENGTH()

```
ival = LENGTH( array )
ival = LENGTH( string-expression )
```

If the argument is an array name, returns the number of elements in the array. If the argument is a string, returns the number of characters in the string.

LOADED()

```
bval = LOADED( program-name )
```

Returns a Boolean value to say if a given program is loaded in memory or not. The program is identified by a string expression that must match the **PROGRAM** name.

LOCATE()

```
ival = LOCATE( value, array )
```

Returns the location of a given value in an array. If the first parameter is an array or the second parameter is not an array, then the index value returned is -1 to indicate a parameter error. If the value cannot be found in the array, then a zero is returned to indicate "not found." If the value is found in the array (by exact match of both data type and value), then the index into the array is returned.

LOCKS()

```
aval = LOCKS()
```

Returns an array describing each JBasic lock in the current process (shared among all users and threads in the same process). The array indicates if the lock is held by any process, if it is held by the current process, the number of locks holds on the lock, and the number of other threads that are waiting on the same lock. This same information can be displayed on the console using the **SHOW LOCKS** command.

The array of records contains the following fields:

Field	Description
“HOLDCOUNT”	The number of current active LOCK operations holding the lock.
“ISLOCKED”	A boolean indicating if the lock is actively locked at this time.
“ISMINE”	A boolean indicating if the lock is owned by the current thread.
“ISZOMBIE”	A boolean indicating if the lock was held by a thread that has exited.
“NAME”	The name of the lock
“OWNER”	The name of the JBasic session that owns the lock.
“WAITCOUNT”	The approximate number of other threads currently blocked while waiting on the lock to become available.

See the documentation on the **LOCK** and **UNLOCK** statements for additional information about how locks are used in thread-intensive programs.

LOWERCASE()

```
sval = LOWERCASE( string )
```

Returns a string with the alphabetic characters in the string converted to lower case if they are upper-case.

MATCHES()

```
bval = ISOBJECT( pattern, string-expression )
```

Returns a Boolean indicator telling if the string expression matches the Regular Expression pattern given as the first argument.

See http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html for additional information on how to formulate and use Regular Expressions in pattern matching.

MAX()

```
val = MAX( arg [, arg...] )
```

Returns the numerically largest value in the argument list, which must contain at least one item.

MEMBER()

```
val = MEMBER( record, "key" )
```

Returns the member of the record identified by the string expression "key". This is used to do indirect access to record members. Consider the following example:

```
x = { name: "Sue", age: 3 }
y = x.name
z = member(x, "name")
```

This code sets y and z to the same value ("Sue"). However, the difference is that the **`MEMBER`**() function call allows the key to be specified by a dynamic expression rather than a fixed identifier. See the description of the function **`MEMBERS`**`()` for how to get a list of the valid keys for a record.

MEMBERS( )

```
array = MEMBERS( record )
```

Returns an array of strings, containing the names of the members of the record passed as a parameter. If the parameter is not a record, then generates an error. Consider the following example.

```
x = { name:"Sue", age: 3 }
y = members( x )
```

This results in the variable Y containing the array `[ "AGE", "NAME" ]`, with each element being a field name (in alphabetical order) from the original record.

MEMORY( )

```
ival = MEMORY( [type-string] )
```

Returns size of memory. The parameter is a string that contains the type of memory value to return. If none is specified, then "USED" is assumed. If an invalid type string value is given, the function returns a value of -1.

The valid memory status types are shown in the following table.

Memory Type	Description
`"FREE"`	The amount of free memory in the process space.
`"USED"`	The amount of memory in use by the process.
`"TOTAL"`	The total memory consumed (used + free)
`"MAX"`	The maximum memory that the process can use.
`"GC"`	The memory after garbage collection has occurred.

MESSAGE( )

```
sval = MESSAGE( status-variable )

sval = MESSAGE( code-string [, parm-string] )
```

Returns a string with a formatted message. If the parameter is a status code variable (such as `SYS$STATUS` or the return code from an **`EXECUTE..RETURNING`** statement) the contents of the status code are used to format a message. Alternatively, you can use a string containing the signal code name and optionally a parameter to be formatted with the message if it has a parameter.

MESSAGES()

```
array = MESSAGES()
```

Returns an array of strings containing the message names of all currently defined messages. These can be used with the **`MESSAGE()`** function to get the message text of all messages, for example.

MIN( )

```
val = MIN( arg [, arg...] )
```

Returns the numerically smallest value in the argument list, which must contain at least one item.

MIXEDCASE( )

```
sval = MIXEDCASE( string )
```

Returns a string of the argument in mixed case. For example, **MIXEDCASE**(`"bob"`) is `"Bob"`; that is, the first character is upper case and the rest of the string is lower case.

MKDIR()

```
bval = MKDIR( path-string [,flag] )
```

Create a directory specified by the *path-string.* Returns true if the directory was created, or false if the directory already existed. By default, any intermediate directories are also created. That is, if the path is “/tmp/a/b/c” then “/tmp/a” and “/tmp/a/b” will be created if needed. If only the exact path given is to be created, pass a *flag* value of false.

MOD()

```
ival = MOD( value, imod )
```

Returns the *value* modulo the *imod* parameter. For example, `MOD(7,2)` returns 1 because that is the remainder of dividing 7 by 2. The *value* parameter can be any numeric expression but is converted to an integer. The *imod* parameter must be an integer value. The result is always an integer. This function has the same as the “%” operator.

MQUOTE()

```
sval = MQUOTE( value )
```

Returns the *value* enclosed in the required characters to make it a *macro string*. See the section on using Macros for more information on a macro string. This function is useful because the literal macro quote characters cannot be expressed as part of program source code, so a program that writes out a macro template will often use this function.

NUMBER( )

```
val = NUMBER( string )
```

Returns a numeric value that is the contents of the string expression parsed as a number. For example, **`NUMBER`**("33") returns the numeric value 33.

OBJECT()

```
rval = OBJECT( var )
```

Returns a record describing the characteristics of the **`OBJECT`** *var*. If the parameter is not an object variable then an error is signaled. Use the **`ISOBJECT()`** function to determine if a given variable is an object (created by the **`NEW()`** function or a **`CLASS`** definition).

OBJECTMETHODS()

```
aval = OBJECT( var )
```

Returns an array describing the methods available for the Java object *var*. Each element of the array is a fully-qualified string description of each method supported by the object. If the parameter is not a Java object variable then an error is signaled. You can see if an object is a Java object by first using the **`ISOBJECT()`** call to see if it is an object, then looking at the **`OBJECT()`** data for it to see if there is a field named `ISJAVA` which will be set to true.

OCTAL()

```
sval = OCTAL( integer )
ival = OCTAL( string )
```

The first form returns a string containing the integer value converted to octal notation. This function is the same as calling `RADIX()` with a radix value of 16. In the second case, if the value is a string the value is converted to a decimal number and returned as the result.

PAD( )

```
sval = PAD( string, count )
```

Returns a string of the argument, padded with blanks to ensure that it is 'count' bytes long. If 'count' is a negative number, padding is done the left; if 'count' is a positive number, then padding is done to the right.

PATHNAME()

```
sval = PATHNAME( file-name-string )
```

Returns a string containing the path component of a file name string. For example, returns "/Users/tom" for the file name string "/Users/tom/myprogram.jbasic".

PROGRAM()

```
sval = PROGRAM( program-name-string )
```

Returns a record containing information about the program identified using the string argument. The resulting record contains the fields described in the following table.

Type	Description
`ACTIVE`	A Boolean that indicates if the program is currently running or not. This can include the current program and the program(s) that called it.
`BYTECODES`	The count of instructions that make up the program, if it has been linked.
`COUNT`	The number of times this program has been executed in this thread.
`HASDATA`	A Boolean that is true if the program has any DATA statements.
`LINES`	An array of strings for each source line in the program. If the program is protected, then this array is empty.
`LINKED`	A Boolean that is true if the program has been linked.
`LOOPS`	A count of the active nested loops, such as **`FOR..NEXT`** loops in the program.
`MODIFIED`	A Boolean that indicates if the program has been modified since it was loaded.
`NAME`	The name of the program.
`PROTECTED`	A Boolean that is true if the program is proteced.
`STATICTYPES`	A Boolean that indicates if the program automatically assumes static data type definitions based on the last character of variable names.
`USER`	A Boolean that indicates if the program was created by the user versus loaded from the JBasic jar file.

RADIX()

```
sval = RADIX( ival, rval )
ival = RADIX( string, integer )
```

The first form returns a string containing the integer value converted to the given radix notation. For example, to convert the integer in *ival* to a binary string, use an *rval* of 2. To convert it to a hexadecimal value, use an *rval* of 16. The second form processes a string containing a value expressed in the given integer radix, and returns it as a decimal integer. So `RADIX("101", 2)` returns a value of 5, which is the decimal integer value of the binary value 101.

REPEAT( )

```
sval = REPEAT( string, count )
```

Returns a string containing the first argument repeated as many times as specified by the second numeric argument. The string can be a single character or a longer string. If the string is empty or count is less than one, then an empty string is returned.

RANDOM( )

```
dval = RANDOM( )
dval = RANDOM( maxval )
dval = RANDOM( minval, maxval )
```

Generates a floating point random number. If no arguments are given, then a number x, such that 0.0 <= x < 1.0 is generated. If one argument is given, then x is between 0.0 and that given value. If two arguments are given, they are the minimum and maximum values returned by the function.

RANDOMLIST( )

```
array = RANDOMLIST( max )
array = RANDOMLIST( min, max )
```

Generate an array of integers sorted in a random order. If one argument is given, then the list contains integers between 1 and that maximum value. If two arguments are given, the array contains integers between the minimum and maximum values. In a **`RANDOMLIST`**() array, no integer appears more than once and all possible integers in the range are represented.

For example, consider the statement `cards = RANDOMLIST(52).` This will generate an array named `cards[]` that has 52 elements numbered from 1 to 52, in random order. This can be used to create a shuffled deck of cards for a game, for example.

RIGHT( )

```
sval = RIGHT( string, count )
aval = RIGHT( array, count)
```

Returns a string containing the 'count' right-most characters from the string argument. If count is less than one, then an empty string is returned.

In the second form, the operation can be performed on an array. In this case, the result is an array that contains the last *count* elements first parameter, which must be an array. For example, the expression `RIGHT([“T”, 3, 8.2], 2)` results in the array `[3, 8.2]`

SECONDS( )

```
dval = SECONDS()
```

Returns a floating point value that is the number of seconds since JBasic was started, accurate to about a millisecond. This can be used to time events, etc.

STRTOARRAY( )

```
array = STRTOARRAY( string )
```

Returns an array, where each element of the array is a single character from the string. The array length will be the same as the length of the string argument.

SUBSTRING( )

```
sval = SUBSTRING( string, start, end )
aval = SUBSTRING( array, start, end )
```

Returns a string value that is a substring of the first argument, starting with the 'start' character and ending with the 'end' character positions. If end-start is less than one, an empty string is returned. If start is less than one or end is greater than the length of the string, then one and string length are assumed.

In the second form, the operation can be performed on an array. In this case, the result is an array that contains elements *start* through *end* of the first parameter, which must be an array. For example, the expression `SUBSTR(["T", 3, 8.2], 2, 3)` results in the array `[3, 8.2]`

SUM( )

```
val = SUM( arg [, arg...] )
```

Returns the numeric sum of the arguments. Differs from **TOTAL**() in that it cannot handle string arguments.

SYMBOL( )

```
rval = SYMBOL( "variable" [, "table"] )
```

Returns a structure that describes the symbol named in the first string parameter. If the table parameter is given, it is a string that contains the name of the table to search for the symbol; if omitted then the search begins in the local table. The resulting record contains the following fields:

Type	Description
`LENGTH`	The length of the value. For an array or record, this is the number of elements in the value; for any other type it is the length in characters of the value when printed.
`NAME`	The name of the symbol
`TABLE`	The symbol table that contains the symbol
`TYPE`	An integer value indicating the type of the value
`TYPENAME`	A string value indicating the type of the value
`VALUE`	A string containing the formatted value suitable for printing.

SYMBOLS( )

```
avalue = SYMBOLS( ["table"] )
```

Returns an array that contains the names of all the symbols in the given table. If the table string parameter is missing, then the local table is assumed. The resulting array is not guaranteed to be in any particular order. Use the **`SYMBOL()`** function to gather information about each symbol represented in the array.

TABLES( )

```
avalue = TABLES()
```

Returns an array that contains the names of all the symbol tables, in the order of most-local to most-global. That is, the first element in the array is always the local symbol table being used to execute the function. This is followed by any nested tables and then the global tables (usually "Global", "Constants", and "Root"). These names can be used with the `SYMBOLS()` function to get a list of symbols in each table.

TOTAL( )

```
sum = TOTAL( arg [, arg...] )
```

Returns the numerical sum of its arguments if they are numeric, or a concatenation of the arguments if they are strings. This differs from **`SUM()`** in that **`SUM()`** can only operate on integer or double numeric values.

TYPE( )

```
sval = TYPE( expression )
```

Returns an string value that describes the data type of the expression. If the expression is a single variable, returns the type of that variable. Return types are "`BOOLEAN`", "`INTEGER`", "`STRING`", "`DOUBLE`", "`RECORD`", or "`ARRAY`".

TYPECHK( )

```
bval = TYPE( expression, descriptor-string )
```

Returns a boolean value indicating if the expression matches the descriptor string. For simple scalar types, this is the same string as returned by the **`TYPE()`** function. However, the descriptor string can contain array and record definitions as well, such that you can compare the type of a complex operator. For example, `TYPECHK( EMPDATA, "{NAME:STRING, AGE:INTEGER,SALARY:NUMBER}"` requires that the data value be a record with three members `NAME`, `AGE`, and `SALARY`. The types of `NAME` and `AGE` are specific scalar data types. The `SALARY` field can be either an **`INTEGER`** or a **`DOUBLE`**. The descriptor string can be arbitrarily complex; arrays must have the same number of elements and element types as the descriptor, and records must have the same members to be considered a match. The special type **`ANY`** will match any type.

This function is particularly helpful when used in a user-written function that wants to verify that all the arguments are the correct type. When a user-written function is called, all the function arguments are in the array `$ARGS`. So the code can use the **`TYPECHK()`** function to determine if the members of the `$ARGS` array are of the correct type to determine if the function arguments were correctly expressed.

UPPERCASE( )

```
sval = UPPERCASE( string )
```

Returns a string with each alphabetic character in the string converted to uppercase.

XML( )

```
sval = XML( value [, formatted] )
```

Converts any JBasic value into an XML representation in a string. This XML value can be transmitted to another instance of JBasic and reconstructed using the `XMLPARSE()` function to convert back to a value. See the section on XML earlier in this User’s Guide for more information.

If the optional second parameter is given, it must be a Boolean flag indicating if the XML is formatted with line breaks and indentation to make it more readable. The default is **`true`**.

XMLPARSE( )

```
value = XMLPARSE( string [, root-tag ])
```

Returns a JBasic value that contains the information encoded in the XML string. A runtime error occurs if the XML is not valid. See the section on XML earlier in this User’s Guide for more information. The default root tag that the function searches for to convert the JBasic value is the tag `<Value>`. You can specify a different name than “`Value`” by passing it as the optional second parameter of the function call. The tag name is not case-sensitive, and *must not* include the “<>” characters.

This page intentionally left blank.

BASIC Compatibility

Every version of BASIC has some implementation-specific features or choices made about how to implement the BASIC programming language. And that language has evolved considerably over time from its relatively simple beginnings at Dartmouth College to the modern-day graphically intensive versions of the language.

It would be difficult if not impossible to catalog the differences between JBasic and every other dialect or variant. However, there are a few common stumbling blocks to converting programs from one version of BASIC to another, and this section will point out the issues you're most likely to encounter when moving programs to JBasic.

Expressions and Variables

Array Notation

In JBasic, an array subscript must always be identified by square bracket characters "[" and "]". For example, an array reference to the third element of an array of names might be `NAMES[3]`. This distinction is important to differentiate an array reference from a function call, which always uses parenthesis. So a function call `NAMES(3)` is always distinct from an array references `NAMES[3]`.

This is different than many dialects of BASIC which use the parenthesis for array references. Converting a program from other dialects to JBasic may involve having to change all array references from parenthesis to brackets.

Variable Typing

In JBasic, by default a variable's type is dynamic. That is, a variable's type is based on whatever value it currently contains. Over the life of a running program, a variable may contain more than one type of data. Additionally, JBasic attempts to convert data from one type to another automatically when needed. For example, an IF statement requires a Boolean expression that yields true or false, but any numeric expression can be used since it can be converted to a zero/non-zero value and then to a true Boolean value.

By contrast, most dialects of BASIC assign one value type to a variable for the life of the program. This static (or unchanging) typing can be used in JBasic, but it is not the default. Additionally, in many dialects of BASIC, the variable name's last character can be used to declare the variable type. For example, going all the way back to the earliest versions of BASIC, a dollar sign "$" character means that the variable is always a string value.

You can use the **`DEFINE()`** option of a **`PROGRAM`** statement to enable static data types for that program. In that case, the first type a variable receives becomes the type the variable retains until the program terminates or the variable is explicitly cleared from memory with a **`CLEAR`** statement. Additionally, when strong typing is enabled, the trailing character of a variable name is used to determine its initial type if it is not explicitly declared with a given type using a statement like **`STRING`** or **`INTEGER`**, or in a **`DIM`** statement.

Defining Variables

As discussed above, you can use dynamic or static data typing for variables to determine what the variable type is. You can also use declaration statements like **STRING**, **BOOLEAN**, **DOUBLE**, or **INTEGER** to define variables and optionally give them initial values. The **DIM** statement can also be used to define both arrays and scalar values with specific types.

Other dialects of BASIC may have different statements for this purpose. Additionally, some dialects allow a statement that defines a range of variables names to have a specific type. For example, some dialects allow DEFSTR to define a range of variables that are always assumed to be **STRING** variables, regardless of name suffix.

Functions

JBasic considers four possible ways you can define a function. You can have a standalone program that acts as a function based on using the **FUNCTION** statement as the first statement of the program. You can use the **SUB** statement to define code within a program to be used either as a function or a subroutine. You can use the **DEFFN** statement to define a statement function which is a single expression that is encoded as a function. Finally, you can use built-in functions that are provided by the JBasic runtime.

Other dialects have somewhat different mechanisms for defining a function. For example, they may allow a statement function using a different syntax like

```
DEF FNL(OFFSET) = CVL(MID$(BUFFER$, OFFSET, 4))
```

JBasic allows relatively relaxed function argument standards, on the assumption that the programmer can decide if this flexibility allows a function to perform slightly different actions based on the type of the arguments, etc. Additionally, JBasic allows varying length argument lists using the ellipsis "..." notation. These features may not be supported in other BASIC dialects. When writing a function in JBasic, you can always examine the $ARGS array in the local symbol table of every function. This contains each of the arguments expressed as an array that you can inspect or use.

Additionally, some dialects of BASIC assume that the function result will be returned in a variable with the same name as the function. For example,

```
FUN DOUBLE(X)
DOUBLE = X * 2
RETURN
```

In this example, the variable DOUBLE is the same as the function name, and is where the result must be stored. In these dialects, the value may be stored at any time during the function code, and is passed back to the caller whenever a **RETURN** statement is executed. By contrast, in JBasic, the value to be returned must be explicitly given as in the **RETURN** statement that exits from the function;

```
FUN DOUBLE(X)
RETURN X * 2
```

Control Statements

Multi-line IF statements

In JBasic, the **IF** statement can be all expressed on one line, or can span multiple lines. When expressed as a multi-line **IF** statement, it has a specific syntax. The **IF** statement itself must have a trailing **THEN** keyword with no statement following the **THEN**. If an **ELSE** block is required, the **ELSE** keyword must be on a line by itself. And the end of the mutli-line **IF** statement block is marked with an **END IF** statement.

By contrast, some dialects of BASIC use slightly different syntax to define a multi-line **IF** statement. Some dialects use an **ENDIF** statement to terminate the multi-statement block, rather than the two-word statement **END IF**. In another example, some advanced versions of BASIC have an **ELSEIF** statement which has the special meaning of defining both the "false" condition for the initial test and a new secondary condition for the next block of code. This creates a "cascading **IF** statement" without having to use the kind of nesting that would be required in JBasic. For example, in some dialects you can use:

```
IF SALARY < 5 THEN
    PAY = SALARY * HOURS * 1.2
ELSEIF SALARY < 10 THEN
    PAY = SALARY * HOURS * 1.1
ELSE
    PAY = SALARY * HOURS
END IF
```

Because JBasic does not have an **ELSEIF** construct, the same code would have to be expressed this way in JBasic. Because there is no **ELSEIF** statement, a second level of nesting is required, along with an additional **END IF** statement to balance the **IF** statements.

```
IF SALARY < 5 THEN
    PAY = SALARY * HOURS * 1.2
ELSE
    IF SALARY < 10 THEN
        PAY = SALARY * HOURS * 1.1
    ELSE
        PAY = SALARY * HOURS
    END IF
END IF
```

DO WHILE and DO UNTIL

JBasic implements most modern looping constructs. One difference is that when the condition is put at the end of the loop body, the keyword **LOOP** is optional in JBasic; you can directly specify the **WHILE** or **UNTIL** clauses without the **LOOP** keyword if you wish.

Additionally, some variations of BASIC have an alternate version of the **WHILE** loop using the **WHILE** keyword and the **WEND** keyword to identify the starting and ending statements of the condition. You can change the **WHILE**..**WEND** block to a **DO WHILE**..**LOOP** block and it should have the same meaning.

CONTINUE LOOP and END LOOP

In JBasic, the **CONTINUE LOOP** and **END LOOP** statements are used to transfer control back to the top of the loop body or out of the loop body, respectively. These statements can be used within a **FOR**..**NEXT** loop or any of the **DO**..**LOOP** constructs.

By contrast, some versions of JBasic have different statements for achieving the same function. For example, some dialects use the **EXIT LOOP** or **EXIT FOR** statement to exit out of the body of a loop. The keyword after **EXIT** determines the kind of loop to be exited. In either case, substitute this with an **END LOOP** statement in JBasic.

A few versions of BASIC (particularly those that are compiled) are able to detect when a **GOTO** statement transfers control out of a loop, and treats it as an **END LOOP** operation, including discarding any runtime information about the loop. JBasic does not have this ability. If you use a **GOTO** to exit out of a loop and then attempt to re-execute the loop, an error will occur because JBasic will think you have duplicated active index variables. For example, the following code will not work in JBasic:

```
START:  FOR I = 1 TO 10
           PRINT “Index = “; I
           IF I = 5 THEN GOTO START
        NEXT I
```

The **GOTO** statement will not correctly discard the pending **FOR**..**NEXT** loop, but will instead attempt to create a new loop with the same index variable as an active loop, which is a runtime error. Change the code to use **CONTINUE LOOP** to iterate to the next loop of the **FOR**..**NEXT** block, or **END LOOP** to exit from the bottom of the loop (after the **NEXT** statement) where execution can be redirected back to the **FOR** statement if needed.

Binary File I/O

FIELD statements

In JBasic, the **FIELD** statement is used to identify the individual data types and sizes in a binary file record. The **FIELD** statement can either store the definition in a variable (actually a JBasic *record* with a specific data structure) or it can bind it directly to an open file. When bound to an open file, the binary record definition is automatically used with **GET** and **PUT** statements that do not specify data or record information explicitly. There can be only one **FIELD** definition linked to a file at a time; the last **FIELD** statement executed defines the record definition used. When stored in a record variable, the **FIELD** definition can be referenced explicitly by the **GET** and **PUT** statements, or can be linked later to a file with the **FIELD**..**USING** statement.

In some modern versions of BASIC, the record definition is created as a Type instead. You will have to translate these Type definitions into **FIELD** statements manually. Most of the data types are supported in both languages, though some versions of **BASIC** refer to an **INTEGER(4)** as a **LONG** and **INTEGER(2)** as a **SHORT**.

Finally, JBasic supports a **VARYING STRING** and a **UNICODE STRING** data type that are not generally supported by other dialects of BASIC. Do not use these in a program where you expect to port your program to other dialects if you can help it. Additionally, **VARYING STRING** creates a data type on disk that is not readable by other programs without understanding the internal representation of a varying length string in JBasic. Avoid this type if you plan on sharing data with other programs.

Binary Data versus Strings

In JBasic, the **FIELD** statement specifies the data type for each individual data item in the record. A **GET** or **PUT** statement using that **FIELD** definition uses this information to determine what data types to use to read or write the data values to or from the binary file.

By contrast, some older versions of BASIC only specify the size of the field, and all data is stored in string variables. It becomes the responsibility of the program to then read the binary data stored in the string variables by converting them as appropriate to integers, floating point values, etc. You will have to change such programs to directly specify the data types in the **FIELD** statement.

This page intentionally left blank.

MULTIUSER Mode

Normally, JBasic runs on behalf of a single user - the one who invoked JBasic from a command line, or perhaps ran a program that uses JBasic as its scripting language. In this mode (referred to as SINGLE USER mode) there are no special considerations for user identity, permissions, or control of file access. These are instead managed by the operating system on behalf of the user, as they are for any running program.

JBasic also supports a mode where multiple users are connected to the same instance of JBasic. This is called MULTIUSER mode. In this mode, there is a single JBasic program running, which accepts connections via a Telnet-style communication protocol, and creates a thread for each incoming connection. Each thread has its own JBasic instance, but they are managed more like a tree of threads (see the documentation on **`CALL`** for information on threads) than as individual connections.

These individual users have a user identity that is not the identity they have on their connection computer but rather the identity they log in with when connection to the MULTIUSER server. These identities are used to manage the location where the user is allowed to read and write files or save programs, and what permissions they have (including do they even have permission to read files, for example).

This mode is intended to support/host a student environment where multiple students may wish to use JBasic under the control of an instructor who places limits on their ability to interact with the hosting computer that is running the actual JBasic server. This mode is not intended for conventional use, because there are significant limits placed on the remote users who connect to a server.

Preparing for MULTIUSER Mode

To run a multiuser user, you need to decide what port the users will all connect on to make the initial contact with the server, and you must also create a user database that describes the identity, permissions, and file system areas that the users will have access to when connecting to your server. Finally you must prepare the directory areas for student use.

The port selection defaults to port number 6100. That is, this is the port number that students will use by default to connect to a JBasic server on your host computer. However, only one server at a time can use any given port number, so if you have multiple servers (for example, one server for each class) then each one must be assigned a unique port number. Additionally, these numbers must not collide with any other well-known port numbers on your system. If in doubt, contact your system or network administrator for suggested port numbers or ranges to use. The global variable `SYS$PORT` must be defined on each server instance of JBasic before the server is started with the **`SERVER START`** command.

User data entry can be done in one of two ways. You can create or edit a text file that contains the user data definitions, or you can use the **`SERVER`** command to create or modify each user identity record. The easiest way to do this is often to use the **`SERVER ADD`** command to create one or two records and then save the database to a text file that you can then edit (using the existing records as a template). You may also wish to consider writing a JBasic program that can help you create the user data programatically.

The server database is an XML file that describes the list of logical names and the characteristics of each user. Because the XML data can be complex, the information in the file is most easily managed using the various **`SERVER`** commands documented in this section.

Logical Names

Logical names are a mechanism for abstracting real physical locations on the computer where the server runs, and presenting a set of locations to remote users that are independent of those real, physical file and directory paths. JBasic allows the administrator to create logical names that identify locations in the host computer's real physical file system that are then hidden from the users, who can only reference those locations by logical name.

Logical names are most often used to identify temporary space where students can write data or the location where the default workspace files are loaded from for the users. The workspace file name itself contains the user's name by default so all workspace files can go in the same directory identified by a logical name. If the logical name begins with an underscore "_" character, then it cannot be viewed by the **`FILES`** command or otherwise inspected by the remote user; only JBasic can load files from these locations.

See the **`SERVER DEFINE`** and **`SHOW SERVER LOGICAL NAMES`** commands for defining or viewing logical names in the following sections.

Home Directories

When students log into the server, they will probably need to have a place to read and write files as well as to save JBasic programs they write. Each student is typically given a different directory area in which to work.

This is set in the user record and should not be created using a logical name. Each user's file activity (both loading and saving programs as well as files read or written by their user programs) are automatically constrained to the home directory of the user. This prevents users from being able to access each others files, or files outside the defined area of the file system.

Do not specify a home directory that is also used for any other purpose.

For example, do not use your own home directory as this will mean that your students can access (read and write or delete) your files. It is recommended that a subdirectory of your computer's temporary space (/tmp or c:\temp, for example) be used for student file system space. When the **`SERVER START`** command is issued, the user home directories are created if they do not already exist.

Functions

There are a few JBasic functions that are designed to assist running programs in participating successfully in a multiuser environment. These are listed below:

PERMISSION()

```
aval = PERMISSION()
   or
bval = PERMISSION("permission-name")
```

If no argument is given, this returns an array of strings containing the names of the permissions granted to the current user. If not in MULTIUSER mode, this will always return an empty array; use the `SYS$MODE` system variable to determine if the function is meaningful for the current mode. If a permission name is given as a string argument, the function returns a boolean true or false value indicating if the current user has the given permission. If not in MULTIUSER mode, this always returns true.

USERS()

```
aval = USERS()
```

Returns an array containing records describing each of the users currently known to the MULTIUSER server. If not in MULTIUSER mode, this returns an empty array. The fields in the resulting record array are:

Permission	Function
`ACCOUNT`	The account name for this user, or an empty string if none given.
`ACTIVE`	A boolean flag indicating if this user is currently logged on.
`LOGINCOUNT`	The number of times this user has successfully logged into the server.
`NAME`	The descriptive name for this user, or an empty string if none given.
`USER`	The user name of this user record.
`WORKSPACE`	The name of the default workspace file for this user.

Controlling the Server

The subsequent sections of this manual will describe the **`SERVER`** and **`SHOW SERVER`** commands in detail. These commands are used to manage the user database, start and stop the server, and display the status of the server and its users. In general, the intent is that the instructor or other administrator starts JBasic, loads the user data, and starts the server mode on behalf of the students. As long as the JBasic server is running, the students have access.

It should be noted that the user database exists in the memory of the server instance of JBasic. You can modify the user characteristics, but you must save them to disk to persist them to be used in the future. See the **`SERVER LOAD`** and **`SERVER SAVE`** commands for more information.

Previous versions of JBasic used the **`SET SERVER`** command to change the state of the server. This created awkward syntax. Starting with release 2.4, these commands are given with the verb **`SERVER.`** For compatibility, the previous **`SET SERVER`** syntax will be supported, but will no longer be documented.

SET PASSWORD

This command is used by a user who is connected to a multi-user server, to change their own password entry in the user database.

```
    SET PASSWORD “my$secret33”
```

The password can be any string expression; in this example, the command sets the password to the string “`my$secret33`” in the user database for the current user.

This command cannot be issued except from a remote user session. The password data is encrypted and stored in the user database. The user must have the `ADMIN_USER` or `PASSWORD` privilege to be able to issue this command.

SERVER ADD USER

The **SERVER ADD** command is used to add a new user to the current user database. The database exists in memory in the JBasic server, and is populated either by a **SERVER LOAD** command or by one or more **SERVER ADD** commands.

```
SERVER ADD USER “SDAVIS” PASSWORD “k234”
   ACCOUNT “CS11”
   HOME “/tmp/sdavis/”
   WORKSPACE “workspace-sdavis.jbasic”
```

This command creates a new user “SDAVIS” and assigns an initial password of “k234”. The user has an account of “CS11” that might be used to designate a class, for example. The home directory for the user is given as well. The workspace defines a file that contains the user’s default workspace that is loaded when they log in, and can be saved with a **SAVE WORKSPACE** command.

Note that while the home directory is “`/tmp/sdavis`”, the user themselves will not see that as part of any file name or directory operation. As far as the user is concerned, they have sole use of the file system for their files, but are in fact actually only seeing files and directories located within “`/tmp/sdavis`” in this example.

The workspace is not located in the user’s home directory, but is a file name (or path and file name) in the server’s process context. That is, the workspace file is loaded and saved in the current directory of the server itself in the above example. This allows the instructor to keep the workspace files separate from the data or individual files that a student might create or use. The user must have `FILE_IO` privilege to be able to save a file or program in their home directory, but they can issue a **SAVE WORKSPACE** at any time to write the current user programs to the workspace even if they have no permissions. This lets the instructor have students “turn in their assignments” by saving to a workspace that can be inspected/graded by the instructor without having to locate files in the user’s account areas.

The password is specified in plain text in this command, but is immediately converted to a hashed value in memory and saved in the hashed form when written to disk. So the user must use the password “k123” to log in, but no one who sees the password file will know what the actual password is. If the user is suitable privileged, they will be able to change the password.

The only required keywords in the **SERVER ADD** command are the **USER** and the **PASSWORD**. All others will be generated as default values if needed.

SERVER DEFINE

The **`SERVER DEFINE`** command is used create a logical name. This logical name can be used by remote users to reference physical file locations by abstract names.

```
SERVER DEFINE TMP = “/tmp/jbasic/userdata”
```

In this example, a logical name TMP is created that points to a location in the file system. The remote users cannot reference this location directly themselves, but they can reference it by using the logical name TMP in their file names. For example,

```
OPEN OUTPUT FILE “TMP::MYDATA.TXT” AS #1
```

The user of the logical name (identified by the “::” characters in the path name) tells JBasic to create this file in the temporary location identified above. The resulting path “/tmp/jbasic/userdata/MYDATA.TXT” is used for the actual physical file, even though this is outside the user’s home directory area. This is the only way that a user can access files outside their home directory, so logical names should never be created that reference important file locations on the host system.

SERVER DELETE USER

The **SERVER DELETE USER** command is used to delete a user record from the database. The database exists in memory in the JBasic server, and is populated either by a **SERVER LOAD** command or by one or more **SERVER ADD** commands. Because of this, the **SERVER DELETE USER** function must be followed by a **SERVER SAVE** command to make the change permanent on disk.

```
SERVER DELETE USER "SDAVIS"
```

This command deletes the user data for the user record "SDAVIS". If the user is still logged in, their session will be terminated by this command, just as if a **SERVER QUIT** command had been given. The user is also prevented from logging in again.

Because this change only happens to the copy of the database in memory, you can exit the server and the record will still exist on disk; you will have only prevented access during the time that the server was running. A subsequent invocation of the server will load the data file again.

SERVER GRANT

The **SERVER GRANT** command is used to grant a permission to a user. These permissions are required to perform operations in JBasic that could consume resources or compromise the security of the JBasic server, and so should be granted only as needed. Note that permissions are only checked when in multi-user mode; programs running in single user mode implicitly have all permissions and can perform any operation in the language. This means that a program may run without error in single user mode, but will result in a permission failure when the same program is run by a user in multi-user mode.

```
SERVER GRANT USER “tom” PERMISSION “ADMIN_SERVER”
```

This command grants the permission “ADMIN_SERVER” to the user named “tom”. It is an error if the user does not exist. Additionally, the permission must be a valid permission name. Here is a table of the permissions that supported by JBasic:

Permission	Function
ADMIN_SERVER	The user is allowed to issue **SERVER** commands that affect the state of the server such as stopping or starting it.
ADMIN_USER	The user is allowed to issue **SERVER** commands that affect individual users or their permissions.
ASM	The user is allowed to use the **ASM** statement to assemble arbitrary bytecode streams. This permission can be used by the user to subvert other permissions and should not be granted except to trusted users.
DIR_IO	The user is allowed to use the **FILES** command and the various functions that read the directory and present information about files in the directory. This privilege is already required to be able to use the KILL statement to delete a file by name.
FILE_IO	The user is allowed to open files using the **OPEN** statement. The user can use **OPEN** to access databases or queues without this permission. This is also required to be able to load or save individual programs using **LOAD** and **SAVE**.
JAVA	The user is allowed to create Java objects via the **NEW()** function or have Java objects created and passed into JBasic via the `addObject()` method.
PASSWORD	The user is allowed to change their password using the **SET PASSWORD** command.

Permission	Function
`SHELL`	The user is allowed to use the **`SYSTEM`** command to execute native commands in the shell. These commands run in the context of the user who started the JBasic server and should only be awarded to trusted users.
`THREADS`	The user is allowed to create threads using the **`CALL`** statement, and to manage threads created by the user programs.

Permissions must be granted before a user logs in to be able to access the permissions; the information about the permissions of a user are copied to the specific user session(s) at the time of log in to the multi-user server. If a user is logged in and additional permissions are granted, new instances of that user will get the new permissions but existing sessions will not.

Granted permissions are stored in the user dictionary in memory; you must execute a **`SERVER SAVE`** command to ensure that these permissions persist in the user database for future JBasic sessions.

See the documentation on the **`SERVER REVOKE`** command for information on how to remove permissions from a user dictionary.

SERVER LOAD

The **`SERVER LOAD`** command loads the user database from a disk file into memory. This is a required step before a server can be started.

```
SERVER LOAD [“filename”]
```

If the file name is not given, then JBasic looks for a file “`JBasic-users.xml`” in the current directory where the JBasic server was started from. If an explicit file name is given, then it will be remembered and used when a **`SERVER SAVE`** command is issued. The file name is relative to the server process, not a user process (even if the command is executed by a remote user).

If you do not explicitly issue a **`SERVER LOAD`** command and there are no users defined when a server start operation is performed, an implicit **`SERVER LOAD`** of the default file is performed automatically. Note that loading users causes them to be added to the database in memory; if there are already users defined they will not be replaced unless a user record with the exact name is found in the input database file.

This command must be performed by the controlling session (the JBasic session launched by the administrator to act as the server) or may be performed by a remote user if they have `ADMIN_USER` privileges.

SERVER MODIFY

The **SERVER MODIFY** command is used to modify an existing user record in the in-memory database. The user must have already been created with a **SERVER ADD** command or by loading from a disk repository of the database.

```
SERVER MODIFY USER “SDAVIS” ACCOUNT “CS23”
```

This command modifies the **ACCOUNT** setting for user “SDAVIS”. You can specify one or more account attributes on the command line. Each attribute consists of a keyword followed by a string constant or string expression.

The following table describes the attributes and their meaning. Also shown is the default value for that attribute if none is given when the user is created or loaded from the disk file.

Attribute	Default	Meaning
PASSWORD	*No default*	The password the user must give to log in.
ACCOUNT	`“DEFAULT”`	A descriptive string about the account, such as class name or department code.
NAME	`“DEFAULT”`	The full name of the user of the account.
HOME	“/tmp/jbasic/*user*”	The home directory in the native file system where the user is allowed to read and write files.
WORKSPACE	“workspace-*user*.jbasic”	The location in the native file system where the user’s **WORKSPACE** is saved.

There is no default for the password because a password must be given when the account is created or loaded from disk, or an error is signaled.

The only user attributes that is not set by this command are the user permissions, which must be handled with the **SERVER GRANT** and **SERVER REVOKE** privileges.

The **SERVER MODIFY** command can be given by the user of the controlling session, or by any remote user with the `ADMIN_USER` privilege.

SERVER QUIT

This command is used to force a remote user session to terminate, as if the user in that session had issued a QUIT command.

```
SERVER QUIT "JBASIC_44"
```

The above command causes the remote user session with instance id `JBASIC_44` to be logged out. Any unsaved work is lost; this command should not be used unless it is necessary to prevent misuse of the server, etc. If the instance ID does not represent an active session, an error is reported.

You can get the instance ID number of each session by using the **`SERVER SHOW SESSIONS`** command. This command may be issued by the controlling session, or by any remote user with `ADMIN_USER` privileges.

SERVER REVOKE

The **SERVER REVOKE** command is used to remove or delete a permission to a user. These permissions are required to perform operations in JBasic that could consume resources or compromise the security of the JBasic server, and so should be granted only as needed.

```
    SERVER REVOKE USER “tom” PERMISSION “ADMIN_SERVER”
```

This command takes away the permission “`ADMIN_SERVER`” from the user named “tom”. It is an error if the user does not exist.

See the documentation on the **SERVER GRANT** command for additional information on the valid permission names and their meaning, and how and when server permissions are assigned to user sessions.

SERVER SAVE

The **SERVER SAVE** command save the user database in memory to a disk file. This is required if the changes made during the current user session are to be made permanent and in effect the next time the server is run.

```
SERVER SAVE ["filename"]
```

If the file name is not given, then JBasic stores the data in the file "`JBasic-users.txt`" in the current directory where the JBasic server was started from. If an explicit file name is given on the last **SERVER LOAD** command issued, then it will be remembered and used when the **SERVER SAVE** command is given. The file name is relative to the server process, not a user process (even if the command is executed by a remote user).

This command may be performed by the controlling session (the JBasic session launched by the administrator to act as the server) or may be performed by a remote user if they have `ADMIN_USER` privileges.

SERVER START

The **SERVER START** command is used to initiate the multi-user mode. Until this is done, the server is not running and will not accept incoming connections.

```
SERVER START [PORT=port-number] [LOGGING=logging-level] [SHELL=program]
```

By default, the server will use port number 6100 to accept new incoming connections. However, you can specify the port number by including it optionally on the command line with the **PORT=** clause, or by setting the global variable `SYS$PORT` to the desired port number before starting the server. Once the server is started, it will use this same port number until it is stopped and restarted; you cannot change the port number of the server without stopping and restarting the server.

Once the server is started, the command prompt changes automatically to "`SERVER>`" to remind you that the current session is the controlling server. You can use the **SERVER SHOW** command to display the current state. When the server is running, users connect to the server using a standard Telnet client of their choice. For example, this session running on a Mac OS X system connects to a remote server:

```
[MyMac:~] macnerd% telnet server.jbasic.net 6100
Trying 192.168.1.33:6100...
Connected to server.jbasic.net.
Escape character is '^]'.

Please log in to JBasic


Username: tom
Password:

JBASIC Version 2.6 (Fri Feb 20 23:22:16 EDT 2009), by Tom Cole

[tom] BASIC>
```

In this example, the user connects to a server on the system named "server.jbasic.net" and specifically identifies port 6100 as the connection port. The user is logged into the server, and must identify himself using the JBasic user database credentials. The password is not echoed to the console. Note that the prompt for the user includes his JBasic user identity.

Once logged in to JBasic, the user has a remote JBasic session and can use any JBasic feature as long as he has the appropriate permissions (see the **SERVER GRANT** command for details).

The logging level set by the **LOGGING=** keyword determines if messages about the state of the server are reported on the main JBasic console. The logging levels are the same as the **SET LOGGING** command.

If the SHELL option is used, this specifies a specific program to be run instead of a command-line shell. When a shell program is given, then no user authentication is required or performed; all users who connect to the port are allowed to run a copy of the program.

```
SERVER START PORT=6101 SHELL=”ELIZA_SHELL”
```

The shell name is any string expression that references the name of a program already in memory. The example shows creating a server to run the program `ELIZA_SHELL`, which is really a shell around the `ELIZA` program included in the default Library in JBasic. The server then allows anonymous server sessions to be created that run code and respond to the user’s request. In these cases, the running program’s input comes from the remote connection and the output is directed back to the remote connection just like a normal remote user connection.

Obviously, in these cases it is very important that the shell program be written in a fashion that prevents the user from accessing resources incorrectly; the resulting program should not all the user to execute file system I/O, etc.

When a shell is created in this way, the user program has all privileges and must carefully ensure that the program’s functions cannot be misused by the remote user.

SERVER STOP

The **`SERVER STOP`** command stops a running multi-user server. All users currently logged in are disconnected, and their work is lost. Do not do this without warning your users.

```
    SERVER STOP
```

Currently there is no mechanism to warn logged in users of an impending shutdown, or to warn the administrator that there are still active users.

When the server stops, the command prompt for the session reverts back to the default prompt, which is normally "`BASIC>`".

If the JBasic session that initially starts the server exits (such as by the user issuing a **`QUIT`** command) then this has the effect of stopping the server and disconnecting the users as well.

SET LOGGING

This command sets the *logging level* for the current instance of JBasic. This determines if error, warning, informational, or debugging messages are sent to the console when events occur within JBasic. In general, these are used by the multi-user mode to notify the primary JBasic console about events that are occurring in the other sessions of the server.

```
SET LOGGING=2
```

This command sets the logging level to 2, which indicates that error, warning, and informational messages are to be sent to the console. The table below shows what kind of messages are generated for each logging level. The default level is zero, which produces only error messages.

Logging Level	Description
0	Only error messages are sent to the console. This is the default logging level.
1	Error and warning messages are sent to the console.
2	Error, warning, and informational messages are sent to the console.
3	Error, warning, informational and debugging messages are sent to the console.

SHOW SERVER

The **SHOW SERVER** command displays information about the state of the multiuser server.

```
BASIC> show server
SERVER=MULTIUSER (CONTROLLING SESSION)
  STARTED=Fri Nov 02 20:24:20 EDT 2007
  PORT=6100
  DATABASE=/Users/tom/workspace/jbasic/JBasic-users.txt
  ACTIVE USERS=TOM
```

This example shows the **SHOW SERVER** output of a running server. The server is running because it is in `MULTUSER` mode. If the mode was `SINGLEUSER` then the server is not running and the JBasic session is only serving a single (current) user. In addition, the message indicates that the session that issued the command is the `CONTROLLING SESSION` - the session that started the server. If this session quits then the server will stop running. The output could also have said `REMOTE SESSION` in which case the session where the **SHOW SERVER** command was executed is a remote session logged into the server but with permission to examine the server state.

Additionally the command shows when the server was started and on what port the server is listening for new connections.

The command also displays the location of the user database (from the **SERVER LOAD** command or the default location) and the current list of active users. In this case, only user “TOM” is currently logged into the server. You can use the **SHOW SERVER USERS** command to get more information about the users.

SHOW SERVER LOGICAL NAMES

The **`SHOW SERVER LOGICAL NAMES`** displays the logical names that are available to the user to specify abstract file path names.

```
SHOW SERVER LOGICAL NAMES
```

See the documentation on logical names and the **`SERVER DEFINE`** command for more information.

SHOW SERVER SESSIONS

The **SHOW SERVER SESSIONS** enumerates the currently-active login sessions. Information is printed about each session, including the instance ID and user name, and information about what that session is currently running.

```
      SHOW SERVER SESSIONS
```

The output of this command can be used to get the instance ID of a session that is to be stopped using the **SERVER QUIT** command.

```
SERVER> show server sessions
  17: TOM               Fri Nov 09 20:09:26 EST 2007
      CURRENT PGM:  AVERAGE
      STMTS EXEC:   6
      BCODE EXEC:   3451
  23: MARY              Fri Nov 09 21:33:49 EST 2007
      CURRENT PGM:  PAYROLL_RUN
      STMTS EXEC:   31
      BCODE EXEC:   12938

2 sessions
```

This example shows two active sessions, with session ID’s of 17 and 23 respectively. The display shows the username and current start time of the session. The output also displays the current programs and an indication of how much work has been done by each session.

SHOW SERVER USERS

The **SHOW SERVER USERS** command displays the current user database. If a user is currently logged in, then it also displays information about the user’s current state.

```
BASIC> show server users
   MARY
     LAST LOGIN    NEVER
     ACCOUNT:      Home
     FULL NAME:    Mary Cole
     HOME DIR:     /private/tmp/jbasic/mary
     WORKSPACE:    workspace-mary.jbasic
     PERMISSIONS:  [ "DIR_IO", "FILE_IO" ]

   TOM
     LAST LOGIN    Fri Nov 02 10:24:17 EDT 2007
     ACCOUNT:      Home
     FULL NAME:    Tom Cole
     HOME DIR:     /private/tmp/jbasic/tom
     WORKSPACE:    workspace-tom.jbasic
     PERMISSIONS:  [ "DIR_IO", "ADMIN_USER", "FILE_IO" ]
```

This sample shows the output of the command when run with two users defined in the database. The display for “MARY” shows that she has not logged in since the server was started, and lists the basic account characteristics from the user database.

The user “`TOM`” has logged in (because he is shown with a `LAST LOGIN`). Use the **SHOW SERVER SESSIONS** to see if the user is stilled logged in, and what he is running.

Inside JBasic

JBasic is written entirely in Java, and is open source. You can use JBasic for any purpose, view the source, or make modifications for your own purposes. This section will describe (in general terms) the structure of JBasic and its internal operation, so you can more easily navigate through the source code or just to better understand its operation.

While developing features for JBasic, or debugging problems encountered running JBasic programs, it is often helpful to understand the internal structure of JBasic, and to be able to use diagnostic tools that support the development of JBasic itself (as opposed to debugging JBasic programs themselves).

This section will (eventually) describe the internals of JBasic, and also describe debugging and diagnostic tools available to the end-user who wants to know more about what is going on "under the hood" in JBasic.

This section will (also eventually) describe how to add additions to the JBasic language without having to modify JBasic itself. Specifically, JBasic allows the addition of statements and functions to the language. This is particularly useful when JBasic is being used as an embedded language processor within another Java program. The extensions allow application-specific statements to be added to the JBasic language and intermingle their operation with the normal JBasic language elements for data manipulation, flow-of-control, etc.

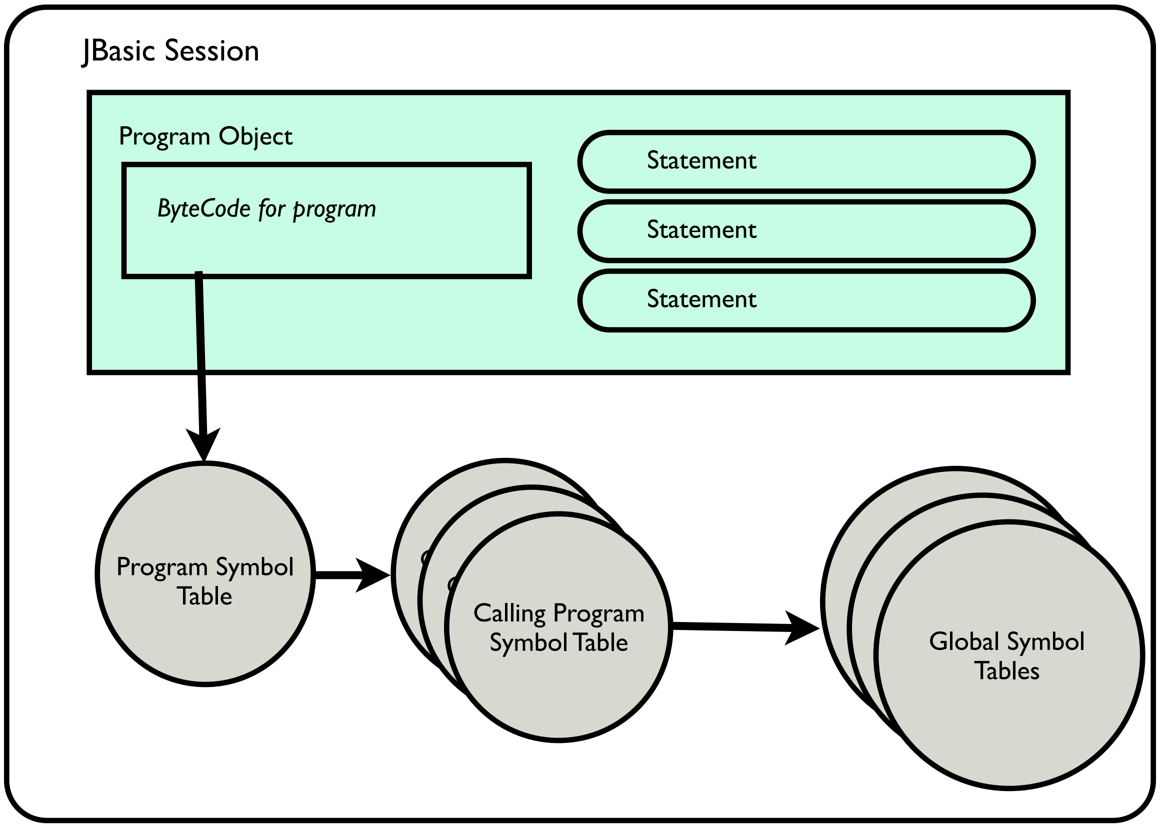
Finally, commands not normally used or needed by the typical JBasic user are described in the following sections.

JBasic Internal Architecture

JBasic has a series of object types that define the major functional areas of the language and the runtime environment. This section will describe them.

The following figure is a rough generalization of the relationship of some of the key object classes. This is followed by a table that describes the most important characteristics of the individual classes that make up a JBasic session.

Note that in this drawing, it shows a single Program with only three Statement objects. This drawing is simplified to illustrate the basic relationships between objects, but is not representative of the typical number of these objects. In normal operation, a JBasic object has many programs, each of which may contain dozens or hundreds of statements.



Here is additional information about each object class in the above figure:

Class	Description
JBasic	This is the primary container for a session. Almost all objects contain a reference pointer to the containing object, which is used to locate shared data structures and object types. There is always at least one JBasic object active for JBasic to be running. When a program uses JBasic internally, it creates a JBasic object for each separate execution context that it needs. When JBasic is run from the command line, the main() method instantiates a JBasic object and then sends console commands to that object to execute. When a **`SHELL`** or **`CALL... AS THREAD`** operation is performed, new JBasic objects are created as children of the initial JBasic object.

Class	Description
Program	This is an executable element of JBasic language code. A program can contain a standard user-written program. It can also implement a **`VERB`** that is a program that is executed automatically when a statement starts with the appropriate verb. It can also be a **`FUNCTION,`** which is a program that is executed as part of expression evaluation and returns a result. Programs contain vectors of statements, which are comprised of the individual lines of user-written program code. Programs also contain the interpreted byte-code that is run when a program is executed.
ByteCode	This is a vector of pseudo-instructions that perform the actual work of a Program. The process of preparing a program for execution consists of compiling each statement into a short string of one or more instructions in a ByteCode object. The program is then linked, which means that the bytecode associated with each statement is concatenated together into a single ByteCode object that is attached to each Program. This ByteCode object contains all the instructions necessary to run the user's program. When a ByteCode stream is executed, a symbol table object is created for it to use for that specific execution context. This allows the same code to be run multiple times (in parallel, recursively, or serially) and each time has a unique symbol table.
SymbolTable	The SymbolTable class describes the symbols available to a given instance of running ByteCode. Each symbol table has a parent symbol table which is the symbol table of the ByteCode that invoked it, or the symbol table owned by the command line shell. When a symbol is accessed, the current symbol table is searched first. If the symbol table cannot be located in the current table, the parent is searched. This continues until all symbol tables have been searched, including the Global Tables. New symbols are always created in the current symbol table (unless specific instructions are given to select parent or global tables). This means that the operation `X = X + 1` may take the value of `X` from a parent table (the calling program's symbol table) and store the new value in the local table.

Class	Description
Global Tables	This is not a specific class, but rather specific instances of the SymbolTable. When a JBasic object initializes, it creates a symbol table named "Global" which is the root of all symbol tables created in that session. The "Global" table has as its root a table called "Constants" which has as its root a table called "Root". The "Root" and "Constants" tables are shared by all instances of JBasic in a single process, including multiple threads. So data stored in "Root" is available to any JBasic program at any time.

Command Execution

This section describes what happens when a command is typed in at a console prompt or passed into the JBasic object by the run() method. In both cases, a string of text describes what the user or calling program wishes to do, and JBasic must act on that command.

The first action is to create a Statement object. The Statement object both maintains the information about the statement (the program text, any executable code) but is also the basic object that gets executed when a single command text is given.

The statement object is passed the text via the store() method, which is responsible for determine how to handle the statement. This is when the statement is compiled, if compilation is possible. This also includes determining if the statement is to be executed directly versus stored in a program.

The statement store() method performs the following steps:

1. If `SYS$RETOKENIZE` is true, the text of the line is reformatted by tokenizing it and then reassembling the line from the discrete tokens. This gives all statements a uniform format and appearance in stored programs and messaging.

2. If the statement begins with a line number, this statement is to be stored in the current program, as opposed to being executed immediately. The line number is removed and a flag set indicating if this is an immediate statement or not.

3. The statement may have a label on it as well (an identifier followed by a ":" character). In this case, remove the label from the program text and store it separately in the statement object. This label may be used later as the designation of a branch.

4. If the statement is an assignment statement (determined by examining the first few tokens of the statement text) then a **`LET`** verb is prepended to the text. For example, this converts the user's statement `X = 3` into the conventional form **`LET`** `X = 3`.

5. The statement verb is compared against the list of aliases in the `SYS$ALIASES` variable. These are verbs that are replaced with another string before parsing occurs. For example, the "**`DIR`**" command is replaced with "**`SHOW PROGRAMS`**", and the "**`?`**" command is replaced with the "**`PRINT`**" command.

6. The statement compiler is called. This attempts to locate a class for handling the verb (the first token of the statement). Statement classes are always of the form *Verb*Statement, which is always a subclass of Statement. The *Verb* in the class name is the command verb with the first letter capitalized.

For example, the class responsible for the **PRINT** verb is called PrintStatement. The search for a class includes the JBasic jar or class path itself, as well as any locations in the SYS$PATH array variable. If a suitable method is located, then its compile() method is called if it exists. This causes the statement-specific method to parse the remainder of the tokens and generate byte code into the statement's ByteCode object handle. If there is no compile() method, then the statement object generates an _EXEC ByteCode that causes the interpreted run() method to be called instead during runtime.

7. If the compile cannot be performed because there is no class for the given verb, then JBasic attempts to locate a program called VERB$*Verb* that will be executed to process the statement. For example, the **DISASM** command is implemented by the program VERB$DISASM. If the program cannot be found, the system attempts to load a program called *verb*.jbasic from the SYS$PATH location(s) to see if it exists but has not been brought into the program space. If a verb program can be found, or loaded and successfully compiled, then it is executed immediately to perform the operation of the statement. The remainder of the tokens from the command text are stored in the $ARGS[] array available to the running verb program.

8. As part of the compile operation, if the statement was to be stored in the current program rather than executed immediately, the store() method arranges for this store operation to be performed. If there is no current program, a **NEW** command is executed on behalf of the user automatically.

9. If the statement is to be executed immediately, the statement then calls the execute() method which runs the statement. This involves calling the ByteCode.run() method to run the byte-code stored in the statement by the compile operation. If this statement is executed via a JBasic.run() method call from another Java program, then the symbol table used to execute the statement is the Global symbol table. In the case of a console or shell statement, a symbol table "Local to Console" or "Local to Shell" has been created and is used as the default table. This table exists as long as the console or shell is running.

10. At the completion of execution, JBasic prompts for another statement or returns to the caller of the JBasic.run() method as appropriate.

Program Execution

Some statements result in the user explicitly or implicitly requesting that a Program object be executed. The most common case is the **RUN** command, but others including just using an expression that calls a JBasic function program or using a verb implemented as a JBasic program can result in the execution of a Program object's compiled code.

To use a simple example, imagine that the **RUN** command is given. The **RUN** command itself is compiled into a byte code stream associated with the command line, and then the byte code is executed (invoking a _CALLP program call).

The program execution is usually accomplished by looking up the appropriate Program object, using the JBasic.findProgram() method that takes a string name for the program to locate. This may be the name stored in the SYS$CURRENT_PROGRAM global variable if the **RUN** command was executed with no options. The JBasic.findProgram() method returns a Program object, and its run() method is called.

When a Program.run() method is called, the Program object prepares itself for execution, and the calls the ByteCode interpreter for its stored program code. The general steps are:

1. The program is linked if it was not previously linked. This operation collects up the byte-code generated for each statement in the program, and concatenates it into a new ByteCode object that is associated with the Program object. It then resolves internal line number and label references to specific byte-code addresses. Finally, if `SYS$OPTIMIZE` is `true`, it calls the Optimizer.optimize() method on the ByteCode that scans over the code looking for ways to make the generated code more efficient, and rewrites the code as needed.

2. If the program has any **`DATA`** statements, the byte-code generated for each data element is collected into a vector associated with the Program object that is used to process the **`READ`** statement operation.

3. A new **`ON`** statement stack element is created if the byte code has any ON statements (implemented using the `_ERROR` byte code). There is a stack (maintained in the JBasic context object) for each execution context that stores any active **`ON`** statement information. This allows each program to have a unique mechanism for handling errors, or to allow the stack to be traversed to locate the parent's **`ON`** statement handlers, etc.

4. A new symbol table is created for this execution of the program. This ensures that each program can have local variables that do not interfere with variables that may be active in the caller's (or the console's) local symbol table.

5. The byte-code for the program is executed by calling its run() method. The ByteCode class is discussed separately.

6. When the byte-code for the program finishes, the local symbol table is destroyed and any **`ON`** statement element for this program execution is removed. The status result of the execution of the byte-code is stored in `SYS$STATUS` and returned to the caller as well. If the caller is another program, then the error (if any) is signaled in that program. If the caller was a console or shell, then the error message (if any) is printed on the console.

Global Symbols

There are system variables and flags that modify the behavior of the JBasic execution environment. Most of these are only useful for debugging problems, since they may render the normal use of JBasic *impossible* when set to non-default values.

Note that these symbols reside in the GLOBAL table, and all are prefixed with the string "SYS$". Any symbol created with this prefix is automatically stored in the GLOBAL symbol table.

Symbol Name	Default	Description
`SYS$AUTOCOMMENT`	`true`	Are comments automatically added to new programs that identify the author and date the program was created?
`SYS$AUTORENUMBER`	`false`	Are programs automatically renumbered when loaded from a workspace? (Regardless of this setting, programs without line numbers are numbered).

Symbol Name	Default	Description
SYS$DEBUG_DEFAULTCMD	"STEP 1"	The default command executed when the user just presses the RETURN key at the debugger prompt. This default steps the program one statement.
SYS$DEBUG_PROMPT	"DBG> "	The default debugger prompt text.
SYS$DISASSEMBLE	false	When true, a **SHOW PROGRAMS** includes the disassembly of the byte-code associated with the program.
SYS$FUNCTION_MAP	[...]	This is an array of records, created during processing of the MAIN program. Each record defines the mapping of a function name from an allowed alternate spelling to the internal function name. For example, **TRM$** is remapped to **TRIM**.
SYS$INDENT	2	The number of spaces to indent **FOR-NEXT**, **DO-WHILE**, and **DO-UNTIL** loops in program listings.
SYS$INPUT_PROMPT	"? "	The default prompt for **LINE INPUT** and **INPUT** statements.
SYS$INSTANCE_ID	1	A unique integer associated with each instance of the JBasic object. This is useful when more than one object is created under control of threads or when used within another Java program.
SYS$INSTANCE_NAME	"JBasic Console"	A text string associated with the JBasic object when it is created.
SYS$ISTHREAD	false	This is true if the current JBasic object was created as part of a **CREATE THREAD** operation, and is running as a child thread of another JBasic instance.
SYS$LABELWIDTH	10	The width of the area in a program listing for label text before the program statements. A smaller number makes for "tighter" listings unless very long (8 character or more) program labels are used.
SYS$LANGUAGE	"EN"	The current language used to translate message text. The default is actually taken from the Java runtime environment when it is available.
SYS$LOAD_LIST	[...]	This is an array describing every object that a load request has been made for. This includes files automatically loaded as verbs as well as **LOAD** commands. The array contains records describing the loaded name, the path it was found in, and a status indicating if the load was successful or not.
SYS$OPTIMIZE	true	If true, the optimizer is run as part of the **LINK** operation to generate more efficient byte-code.

Symbol Name	Default	Description
SYS$PACKAGES	[...]	This is an array of strings which identify the package names that will be searched when locating statement handlers at compile and run times. This is a read-only variable and is modified when the enclosing program calls the JBasic addPackage() method.
SYS$PATH	[...]	This is an array that contains the locations where JBasic should search for language objects. For example, when searching for a statement handler for a verb that has been parsed, this list is used to search for a suitable class. Extending this list allows user-written statement handlers to be added to the language dynamically at runtime.
SYS$PROGRAMS	[...]	This array contains the name of every executable program object (programs, verbs, and functions) active in the current JBasic object.
SYS$PROMPT	“BASIC>”	This is the default command prompt for JBasic.
SYS$RETOKENIZE	true	When true, statements are reformatted based on internal rules for capitalization, spacing, etc.
SYS$SAVEPROMPT	“...”	This is the prompt string that is displayed if the user attempts to quit JBasic but there are unsaved program modifications. If the string is empty, then no prompt is issued.
SYS$SHELL_LEVEL	0	This is the numbers of layers of **SHELL** commands active. A **SHELL** command creates a new instance of JBasic and runs it until it exits, then resumes the current JBasic instance. Unlike a thread, the current JBasic does not continue executing.
SYS$START_TIME	floating point	This floating point number indicates (in seconds) how long it took JBasic to initialize. This does not include the time it takes Java to initialize, but is the time from the creation of the first JBasic instance to the first command prompt or execution of a user program statement.
SYS$STATIC_TYPES	false	When true, variables have static types; that is, they have one type for the life of the variable. This type can be set with a **DIM** statement, or defaults based on variable name. Static types may be required for programs written for some dialects of BASIC but are slower than dynamic (default) data types.

Symbol Name	Default	Description
`SYS$STRUCTURE_POOLING`	`false`	When a program has a large number of structure constants, turning this on results in a single copy of the structure being created, and all other instance of the constant are references to the one copy. This is a performance win if the structure has more than 3 members and is used more than 5 times.
`SYS$THREADS`	`[]`	This is an array of records that describes all the child threads of this JBasic object.
`SYS$TIME_GC`	`true`	When true, the **`TIME`** command initiates a Java garbage collection operation before running the program, to reduce the variability of the memory map when running benchmark programs.
`SYS$TRACE_BYTECODE`	`false`	When true, each byte-code operation that is executed is displayed on the console, along with the current top of the data stack. This is used to trace program execution at the byte-code level, but is very slow and produces a lot of output.

Runtime Statistics

There is a special subcategory of global symbols that are used to hold runtime statistics. These variables all have the prefix "`SYS$$`" and have the special property of being associated with specific Java variables inside the JBasic object hierarchy.

Symbol Name	Description
`SYS$$FCACHE_HITS`	The number of times that a lookup of a function name could be satisfied from the local cache rather than requiring the full Java-based search for a suitable class and method.
`SYS$$FCACHE_TRIES`	The number of times that a function lookup was attempted on the function cache. The ratio of this variable to SYS$$F-CACHE_HITS tells how efficient the cache is.
`SYS$$INSTRUCTIONS_EXECUTED`	This is the number of byte-code instructions executed.
`SYS$$STATEMENTS`	The number of JBasic statements executed.
`SYS$$STATEMENTS_COMPILED`	This is the number of statements that have been successfully compiled into byte-codes.
`SYS$$STATEMENTS_EXECUTED`	This is the number of statements executed as byte-code.
`SYS$$STATEMENTS_INTERPRETED`	This is the number of statements executed that had to be handled interpretively (via a run() method) rather than being compiled into byte-code.

Value Representations

Perhaps the most important internal object class in JBasic is the Value class, which is used to hold all information (numeric, string, array, or record) created by or manipulated by user commands and programs.

- All variables in JBasic are implemented as Value objects, using Symbol Tables.
- Executed ByteCode instructions also manipulate Value objects on a runtime stack.
- Symbol Tables are data structures containing Values that are mapped to names.

An object of type Value can have any supported data type, though it can only have one type at a time. The Value object can be created with a specific type and data value, or it can be created and the type and value set independently.

Value objects have accessor functions to get and set the value. They also have utility methods to format the value for printing, or for converting the value to a different type. And Value objects can be compared for equality by determining if two Value objects represent the same numeric or string value.

ByteCode Execution

All statements are compiled into ByteCode objects, which contain the instructions for how to execute the statement. This portion of the document will describe the nature of ByteCode instructions and the runtime environment they are used in.

Each statement has a ByteCode object, which must be initialized by the compile() method of the appropriate statement handler class within JBasic or a user-supplied class library. If the statement cannot be successfully compiled, the ByteCode is discarded.

In addition, there is a ByteCode object associated with each program that is created by the Linker before a program is run. This is a concatenation of the ByteCode for each statement in the program, with internal references to line numbers and labels converted to pointers to specific ByteCode instructions.

In both cases (individual statements executed as commands, and complete programs), the execution of the ByteCode is essentially the same, and involves calling the run() method of the ByteCode object, passing it a pointer to the current local symbol table.

The ByteCode executes as a stack-oriented pseudo-instruction-set architecture. The ByteCode consists of an array of Instruction objects, each of which defines a single fundamental operation in the JBasic language or specific sub-semantics of JBasic language elements. The Instruction also contains any needed arguments to the instruction. In addition to the information in the instruction, the ByteCode object maintains a data stack that is a simple LIFO stack of Value objects, and a stack pointer indicating how many items are on the stack. The ByteCode also contains an "instruction pointer" that is an integer index into the Instruction vector that tells which instruction is to be executed next.

Each Instruction object contains the following fields:

Object Field	Description
`opCode`	An integer describing which instruction this is to perform. The ByteCode class defines a list of constants for this purpose. For example, `ByteCode._ADDI` is used to represent the `_ADDI` instruction.
`integerValid`	A Boolean that indicates if this instruction has a valid integer operand.
`integerOperand`	The integer value in the instruction, assuming `integerValid` is true.
`doubleValid`	A Boolean that indicates if this instruction has a valid double floating point operand.
`doubleOperand`	The double value in the instruction, assuming `doubleValid` is true.
`stringValid`	A Boolean indicating if this instruction has a valid string operand.
`stringOperand`	The string value of the instruction operand, assuming `stringValid` is true. If there is no string operand, this object field is `null`.

ByteCode programs start with an empty data stack, and begin at the first Instruction found in vector location zero. The opcode field of the instruction is used to locate the execute() method of the appropriate class that supports the instruction. For example, the ByteCode called `_ADDI` (add integer) is implemented in the class `OpADDI` located in the `org.fernwood.jbasic.opcodes` package. All Instruction implementations are in this package.

The instruction method has the responsibility of using the additional operand fields of the Instruction, as well as any data needed from the data stack, to perform its operation. The instruction methods are all void methods, so any value they return must be put back on the data stack. If an error is detected by the instruction, a JBasicException signal is thrown. The ByteCode run() method uses this information to signal an error as appropriate.

In the example of the `_ADDI` instruction, the instruction removes the top Value from the data stack, coerces it to be an integer value, adds the integer operand in the instruction to the value, and pushes the resulting new value back on the stack.

An instruction that changes flow of control from the normal order of just executing the next Instruction in the ByteCode vector modifies the "instruction pointer" to point to the next instruction to be executed as appropriate. For example, the `_BZ` instruction removes the top stack item to determine if it is equal to zero or Boolean false. If so, then it uses the integer operand to set the new instruction pointer, so the next instruction executed is the one described in the `_BZ` instruction.

This page intentionally left blank.

Internal Commands

The remainder of this document contains descriptions of commands that are not generally used by the end-user, but are to support development and debugging of JBasic itself.

Unlike typical language statements, these are more likely to change as needed without regard for compatibility with previous versions of JBasic, or for conformance with other dialects of BASIC. As such, exercise caution in using them directly in programs since their function may change in the future.

ASM

The **ASM** command is used to immediately process text descriptions of bytecodes and store them in a statement for execution. If used in immediate mode (as a statement entered at a command prompt) the assembled bytecodes are executed immediately. When used as a statement in a program, the specified bytecodes are stored as the compiled statement instructions. This allows arbitrary sequences of byte-codes to be generated and tested.

```
100   ASM _INTEGER 3, _ADDI 5, _OUTNL 0
```

This results in the number 8 being printed on the default console, since the statement bytecodes generated will load an integer 3, add an integer 5, and print the result. Each bytecode must be separated by commas, and consists of an opcode value and optionally an integer, double, and string argument. No attempt is made to determine if the bytecode is constructed in a valid fashion; that is determined at runtime when the statement is executed.

There is currently no documentation on individual bytecode operations. You can use the **DISASM** command to see examples of bytecode streams generated by specific JBasic statements. In sand-boxed or multiuser mode you must have the ASM privilege to be able to use this statement.

You can assemble a sequence of bytecodes that have been stored as strings in an array. For example,

```
100 CODE = [ “_INTEGER 3”, “_ADDI 5”, “OUTNL 0” ]
110 ASM USING(CODE)
```

In this case, the value passed to the **ASM USING()** statement must be an array of strings. Each array element is the definition of a single bytecode operation. This array can also be created using the `COMPILE`() function.

COMPILE

This statement is used internally to test byte-code compilation and optimization. The command accepts a complete statement as the rest of the command, and compiles it into byte-code. The resulting byte-code is displayed, and if it can be optimized, the resulting optimization is also displayed. The statement is then executed.

```
COMPILE PRINT "This is a test of "; ME
```

The above will compile the **`PRINT`** statement, display the resulting byte-code, and then execute the statement.

COMPILE()

The **`COMPILE()`** function is used to compile a single statement and generate an array of bytecode definitions that describe the statement. This array can be used with the **`ASM USING()`** statement to execute the bytecode at a later time. For example,

```
100  CODE = COMPILE("X = 33")
110  ASM USING(CODE)
```

The first statement compiles the program statement to assign the value 33 to the variable `X`. At the time of this compilation, the assignment to `X` is not executed, only compiled, and the representation of the bytecodes is stored in the variable `CODE`. Only when the **`ASM USING()`** statement is executed does the assignment to the variable `X` occur.

In the above example, the variable `CODE` contains the array describing the bytecodes, which will be something like `[ "_STMT 0", "_STORINT 3 \"X\"", "_END" ]`

The **`COMPILE()`** function is normally used just for diagnostic purposes, but can be used to determine how to store away encrypted code. For example, you could define a critical statement using the **`COMPILE()`** function and then use the **`XML()`** function to express it as a string and the **`CIPHER()`** function to encode it. The resulting bytecode would be unintelligible to any user or program until the **`DECIPHER()`** and **`XMLPARSE()`** functions where used to restore it to a form that could be executed by **`ASM USING()`**.

DISASM

The **DISASM** command (like the **COMPILE** command) is used internally to develop and debug issues with byte-code compilation. Most (but not all) JBasic statements are compiled into a simpler semantic representation called byte-code, which defines the actual operations required by the JBasic statement given. The **DISASM** command is used to list a *program, function, verb,* or *test,* and also display the byte-code instructions associated with the JBasic statements in the named program unit.

```
DISASM type name
```

Where *type* must be one of **PROGRAM**, **FUNCTION**, **TEST**, or **VERB**. This is followed by *name,* which is the name of the object to be listed. For example,

```
DISASM PROGRAM BENCHMARK
        or
DISASM FUNCTION UPPERCASE
```

If no type or name is given, then the current program is disassembled. If the program has not been run yet, then each individual statement is disassembled. If the program has been run at least once then it has been *linked,* which means all the statements have been combined into a single byte-code stream, and symbolic labels have been converted to specific byte-code locations.

LINK

The **LINK** statement forces the link operation to be performed on the current program. When a program is linked, the code generated for each program statement is assembled into a single byte-code stream and internal labels and statement number references are resolved. Additionally, the optimizer performs a pass over the final program as long as **SET OPTIMIZE** is in effect (it is on by default).

```
    LINK
```

When a program is first loaded from the workspace, it is compiled but not linked. That is, each statement is compiled as it is entered, but the byte-code is stored with each statement object.

When the link is performed, each statement's byte-code is collected up and stored in the program executable byte-code object. This is the byte-code that is run when a program is executed.

When a new statement is added or deleted from the program, the linked byte-code is destroyed. This is because new statements can change the relationship between statements, add labels, or delete code.

When a program is run, if it has not been linked, it is linked *automatically* before execution (this includes functions and subprograms). As such, the **LINK** command is never required to be executed by the user. However, you can use the **LINK** command to force the link operation. This is most useful in combination with the **DISASM** and **UNLINK** commands to view the completed program or to review the effectiveness of the optimizer.

SET

The **SET** command is used to set options that effect the runtime behavior of JBasic. You can use the **SHOW OPTIONS** command to display the current settings.

To set an option, just name it. To turn the option off, prefix it with **NO**. For example,

```
    SET OPTIMIZE          // Enable the optimizing compiler
    SET NOTRACE           // Disable byte code execution tracing
```

You can specify more than one option at a time by separating them by commas:

```
    SET TRACE, PROMPT="BAS> "
```

This illustrates that some options (such as **PROMPT**) accept a value in the command.

There are a few commands that are not reflected in the **SHOW OPTIONS** output. One example is the command that controls the way that source code is formatted.

```
    SET NEW_FORMATTER    // OR USE NONEW_FORMATTER TO TURN OFF
```

When source code is automatically formatted by JBasic, there are two styles of formatting. The old version of the formatter (used in versions prior to JBasic 2.8) included lots of spaces in the output for maximum readability, but caused the source lines to be much longer than needed. Starting in JBasic 2.8, the default formatter uses far fewer spaces, which makes lines of text shorter and denser. For example,

```
   100      MID$( A, B, B + 2) = “XX”   // OLD STYLE

                  ...versus...

   100      MID$(A,B,B+2)=”XX”          // NEW STYLE
```

You can select which formatter is used with the **SET [NO]NEW_FORMATTER**. The default is to use the new format rules. When you change this setting, if there is a current program then it is reformatted using the newly-set format rules.

SHELL

The **SHELL** command is create a new instance of a JBasic object, and run a single command or access a console to accept user input.

```
SHELL [“command”]
```

If the command is not given, then the newly created JBasic object prompts for input. When a console or command is running under the control of a shell, the system variable `SYS$SHELL_LEVEL` is set to a non-zero number, indicating the ‘depth’ of active shells. The first shell created has a level of 1, and if it creates a shell, that will have a depth of 2, etc.

Shells exist until either the command given completes execution, or if no command is given, until a **QUIT** operation is performed within that shell.

While a shell is running, all other program execution of the parent JBasic object is suspended. So if a program uses the **SHELL** command to execute a command or statement, the program stops until the shell command completes.

The advantage of a shell is to create an environment that can be cleaned up simply by completing the command or exiting from the shell. Since the shell has its own global symbol table, you can create a shell to experiment with different runtime settings or diagnostics. When the experiment is over, **QUIT** the shell and you are still in JBasic but have discarded the modified JBasic execution environment.

From an internal point of view, the **SHELL** mechanism is what is used by the **CREATE THREAD** statement, but the shell is started on another thread rather than in the current thread.

The command prompt for a shell is “`SHELL_n>` “ where *n* is the shell level number.

SHOW CALLS

The **SHOW CALLS** command is used to display the current active call stack. This describes which programs have called which other programs or functions.

For example, consider a program FOO that calls a program BAR that calls the user-written function CARDS(). While debugging this set of programs, the user issues **STEP INTO** commands to step into each program as it is called. While in the CARDS() function, a **SHOW CALLS** command is given.

```
DBG> show calls

 1: FUNCTION CARD  150    IF( N < 1 ) OR( N > 52 ) THEN RETURN "<Invalid>"
 2: CALL BAR       220    LET X = CARD( 3 )
 3: RUN FOO        100    CALL BAR

3 stack frames
```

This example shows that the current (#1) item in the call stack is the current invocation of the CARD() function (note that each level indicates how it was invoked, so in this example CARD was called as a function). The statement about to be executed (the *current statement*) is displayed along with its line number.

The second item in the call stack shows the line in the program BAR that invoked the function, and its line number. The third item in the call stack shows the original calling program FOO that invoked the program BAR.

In all there are said to be three stack frames (a frame describes all the contextual information needed for a given invocation of a program, such as the program itself, its symbol table, etc.) The currently executing statement is at the top of the list.

SHOW FILES

The **SHOW FILES** command displays information about the currently open files in JBasic. This describes the external information about the file (such as its name) and the internal information including how it is referenced in user programs and whether it is a system file or not.

Consider the following file open operations:

```
      100   open file "acctdata.txt" for output as #1
      110   open file "usernames.txt" for input as uns
```

See the documentation on the **OPEN** statement for more details about the syntax of this command. If the above statements are active in a program, and a **SHOW FILES** command is issued, the output might look something like this:

```
BASIC> show files

CONSOLE_INPUT   { FILENAME: "%console", MODE: "INPUT", SEQNO: 1, SYSTEM: true }
CONSOLE_OUTPUT  { FILENAME: "%console", MODE: "OUTPUT", SEQNO: 2, SYSTEM: true }
#1              { FILENAME: "acctdata.txt", MODE: "OUTPUT", SEQNO: 4, SYSTEM: false }
UNS             { FILENAME: "usernames.txt", MODE: "INPUT", SEQNO: 5, SYSTEM: false }
```

This lists all files that are currently open and that can be used for input/output operations such as **PRINT** or **LINE INPUT** commands. Each file is described by a name and a record value. In general, the name is an active symbol in the current symbol table, and points to the actual record value displayed here. For example, you can issue a **PRINT** `CONSOLE_INPUT` command to see the description of this file.

The first two files are system files (the **SYSTEM** field value is `true`). This means they were created by JBasic and cannot be closed by a user program. These two files are always available for use by a program, and are the default locations where **PRINT**, **INPUT**, and **LINE INPUT** operations are directed if no explicit **FILE** clause is used on those statements.

The third entry describes a file created with the numeric format (#1, in this case). A symbol name is created on behalf of the user (`__FILE_1`) that identify this file, but it is displayed using the #n convention. The **SYSTEM** attribute is `false`, which means it is a user file that can be closed by a user program.

The fourth entry was opened using the JBasic version of the **OPEN** statement where an identifier is given in the **OPEN** statement that holds the data record describing the open file, and can be passed to any **FILE** clause in other statements such as **LINE INPUT** to acquire input records from that file.

In each file record, there is a **MODE** field. This describes the mode that the file can be used in, and comes from the list supported by the **OPEN** statement, with values such as **INPUT**, **OUTPUT**, or **DATABASE**. The **SEQNO** field is a unique number used to identify each individual open file in the life of the JBasic session; no two files will ever have the same sequence number.

SHOW FUNCTIONS

The **SHOW FUNCTIONS** command is used to display the currently defined runtime functions available for use in a program. An example output is:

```
BASIC> show functions
         ABS()                    ARCCOS()                 ARCSIN()
         ARCTAN()                 ARRAY()                  ASCII()
         BASENAME()               BOOLEAN()                CEILING()
         CHARACTER()              CIPHER()                 COS()
         DATE()                   DECIPHER()               DOUBLE()
         EOD()                    EOF()                    EXISTS()
         EXP()                    EXPRESSION()             EXTENSION()
         FILEPARSE()              FILES()                  FILETYPE()
         FILETYPES()              FLOOR()                  FORMAT()
         GETPOS()                 HEXADECIMAL()            INTEGER()
         ISOBJECT()               LEFT()                   LENGTH()
         LOADED()                 LOCATE()                 LOWERCASE()
         MATCHES()                MAX()                    MEMBER()
         MEMBERS()                MEMORY()                 MESSAGE()
         MESSAGES()               MIN()                    MOD()
         NAN()                    NEW()                    NUMBER()
         OBJECT()                 OCTAL()                  OPENFILES()
         PARSE()                  PATHNAME()               PROGRAM()
         PROPERTIES()             PROPERTY()               QUOTE()
         RADIX()                  RANDOM()                 RANDOMLIST()
         RECORD()                 REPEAT()                 REPLACE()
         RIGHT()                  ROUND()                  SECONDS()
         SIN()                    SIZEOF()                 SORT()
         SQRT()                   STRING()                 SUBSTR()
         SUM()                    SYMBOL()                 SYMBOLS()
         TABLES()                 TAN()                    THREAD()
         THREADS()                TIMECODE()               TOKENIZE()
         TOTAL()                  TRIM()                   TYPE()
         UNIQUENAME()             UNIQUENUMBER()           UPPERCASE()

    USER-WRITTEN FUNCTIONS:
        ARRAYTOSTR()             CARD()                   ENCODE()
        ISOBJECT()               MIXEDCASE()              PAD()
       *PARITY()                 PATH()                   PI()
        STRTOARRAY()             UID()
```

This lists the built-in functions and the “user written” functions, which are functions implemented in JBasic itself. The “*” character indicates that the function was loaded from a user program as opposed to the built-in library of pre-supplied JBasic functions.

You can display a single function by using the **SHOW FUNCTION** command followed by a function name. If the function is implemented as a JBasic program, the function program code is displayed. If the function is a built-in compile-time or run-time function, that information is displayed:

```
BASIC> show function pad

PROGRAM FUNC$PAD

  100          FUNCTION PAD( STR, INTEGER SIZE )
  110
  120          //  PAD( string, length )
  130          //
  140          //  Returns a string with enough blanks to make a string
  150          //  exactly "length" characters long.
  160
  170          IF SIZE = 0 THEN RETURN ""
  180          IF SIZE > 0 THEN
  190            RETURN LEFT( STR || REPEAT( " ", SIZE ), SIZE )
  200            ELSE
  210            LET SIZE = - SIZE
  220            RETURN RIGHT( REPEAT( " ", SIZE ) || STR, SIZE )
  230          END IF
  240          END

BASIC> show function sum
SUM is a compiled-in JBasic function

BASIC> show function xml
XML is a runtime JBasic function
```

SHOW LOCKS

The **SHOW LOCKS** command is used to display the list of known locks available to the current JBasic process. This includes the primary session, all shell sessions, all remote user sessions, and all threads - each of which shares a common lock database, to support locking of critical regions of code.

```
BASIC> show locks
3 locks defined:
  1: *ACTIVE_USER(1/0)
  2:  DB_HEADER (1/2)
  3:  USER_LIST (2/5)
```

The **SHOW LOCKS** command displays all locks. In the above example, there are three locks. The "*" asterisk character indicates that the current thread holds the lock at the time of the display. This is followed by the name of the lock, and lock count information.

The first lock count value shows the number of times the lock has been acquired by the current thread. This number must be zero before the lock can be acquired by another thread. The count is increased if more than one **LOCK** statement is given for the same lock already held by the current process. In the example above, most locks have the default value of one. However, the lock `USER_LIST` has a hold count of two, which means that whatever thread holds the lock has actually locked it twice, and will need to issue an **UNLOCK** statement twice to make the lock available.

The second number shows the number of other threads that are currently waiting on the lock. Each of these represents another thread of execution that is stalled out until the owning thread fully releases the lock. The the case of the lock `ACTIVE_USER` the, the current thread holds the lock but no other thread is waiting on the lock. The lock `DB_HEADER` is held by one thread (not the current thread) and two other threads are waiting to acquire the lock before they can resume execution. The lock `USER_LIST` has five threads waiting for the lock to become available.

See the documentation on the **LOCK** and **UNLOCK** commands for more information on using locks in threaded JBasic programs. Use the **CLEAR LOCK** command to delete locks that are no longer needed.

SHOW MESSAGES

The **SHOW MESSAGES** command displays the current mapping of signal names to message text strings. This includes the messages that are defined by default when JBasic initializes, as well as any new message mappings created by the **MESSAGE** command since the session started.

Here is a sample excerpt of the output of the **SHOW MESSAGES** command:

```
BASIC> show messages
   *BRANCH(EN)      Branch execution
   *NOCOMPILE(EN)   No compilation support for %s statement
   *NOCOMPILE(FR)   Ne peut pas compiler le rapport
   *QUIT(EN)        Quit JBasic
   *RETURN(EN)      Return from program or function
   *STMTADDED(EN)   JBasicStatement added to active program
   *SUCCESS(EN)     No error
   *SUCCESS(ES)     Ning?n error
   *SUCCESS(FR)     Aucune erreur
   *SYSTEM(EN)      SYSTEM process return code %s
   *UNSAVED(EN)     QUIT cancelled
    ARGERR(EN)      Argument list error
    ARGERROR(ES)    Error en par?metros de la funci?n
    ARGNOTGIVEN(EN) Argument not given for %s
    ARGNOTGIVEN(ES) Par?metro no dado para %s
    ARGTYPE(EN)     Argument of wrong type
```

For any given message code, the leading asterisk character ("*") indicates that the message is considered a successful condition and does not cause termination of a program.

Each message code has a language code associated with it. There is an English message string for all codes, and additional message strings may be added for other languages over time. The message code is a two-character string corresponding to the Java language codes. For example,

Code	Language
EN	English
ES	Spanish
FR	French

Finally the text of the individual message is displayed. The "%s" element describes where the substitution string (error message parameter) is inserted into the text of the message, if there is a parameter.

SHOW OPTIONS

The **`SHOW OPTIONS`** command displays the settings of common runtime options that can be modified using the **`SET`** command, also documented in this section on internal commands. For example,

```
BASIC> show options
Options: PROMPT="BASIC> ",
         TOKENIZE,
         OPTIMIZE,
         NOTRACE,
         NOSTATIC_TYPES,
         NOPOOLING
            ...
```

This sample output shows the various settings, which are also reflected in internal system variables:

Option	System Variable	Description
`PROMPT`	`SYS$PROMPT`	The console prompt; the default is "`BASIC>`"
`INPUTPROMPT`	`SYS$INPUT_PROMPT`	The **`INPUT`** and **`LINE INPUT`** prompt; the default is "`? `"
`DEBUGPROMPT`	`SYS$DEBUG_PROMPT`	The Debugger command prompt; default "`DBG> `"
`LANGUAGE`	`SYS$LANGUAGE`	The default language for messaging. The default is the current language setting for Java.
`LABELWIDTH`	`SYS$LABELWIDTH`	The number of spaces reserved in listings for statement labels. The default is ten characters.
`OPTIMIZE`	`SYS$OPTIMIZE`	When true, all statements are processed by an optimizer to improve program performance. The default is true, and should only be turned off when a bug in the optimizer is suspected.
`TOKENIZE`	`SYS$RETOKENIZE`	When true, program statements are reformatted to make uniformly formatted program statements. The default is true.
`TRACE`	`SYS$TRACE_BYTECODE`	When set, all byte code instructions are traced by displaying them to the console, along with the current size of the data stack and the top stack item. This is used to debug internal bytecode errors, and the default setting is false, which displays no trace data.

Option	System Variable	Description
STATIC_TYPES	SYS$STATIC_TYPES	This determines if variables are statically typed for new programs that do not explicitly have a type designation. See the documentation on the **PROGRAM** statement for more information. The default is false which means variables are dynamically typed and can change type at runtime.
AUTOCOMMENT	SYS$AUTOCOMMENT	When true, programs created with the **NEW** statement automatically have a header comment generated for them. When false, a **NEW** program is empty.
AUTORENUMBER	SYS$AUTORENUMBER	When true, programs are automatically renumbered when they are loaded from external files via **LOAD** or **LOAD WORKSPACE** commands.
TIMEGC	SYS$TIME_GC	When true, the **TIME** command first does a "garbage collection" operation in memory to free up unused storage previously allocated to variables, etc. This makes the **TIME** command more consistent between executions because the memory allocation for JBasic is more likely to be comparable between executions.
PROMPTMODE	*none*	When set, JBasic prompts for each command line most command-line programs and tools, using the default PROMPT string. When clear, JBasic only prints a prompt after a program runs or is loaded, which emulates older BASIC user environments.
STATEMENT_TEXT	*none*	Wen set, JBasic stores the text of each program statement in the _STMT bytecode in the compiled program. This makes disassembly easier to read but takes up more storage. When clear, the _STMT bytecode only contains the line number, not the full program text.
NEEDPROMPT	*none*	Indicates that a prompt should be printed in the shell when the current program or command completes.

SHOW PROGRAM

The **`SHOW PROGRAM`** command can be used to display all programs, a specific program, or provide internal diagnostic data about a program.

```
SHOW PROGRAMS
```

The **`SHOW PROGRAMS`** variation displays the currently available programs that are in the workspace. These are listed in alphabetical order, and have an indicator that shows if the program is a user-written program ("*") or a user-written program that has been modified but not saved ("#"). When the user quits JBasic, if there are program that are modified but not saved, the user is warned to use **`SAVE`** or **`SAVE WORKSPACE`** to prevent losing those changes.

```
SHOW PROGRAM MYPGM
```

This displays the text of a specific program named `MYPGM`. This is the same function as the **`LIST`** command, but can operate on a program other than the current program.

```
SHOW PROGRAM(DISASM) MYPGM
```

This variation displays the bytecode disassembly along with the program text. For programs that are not linked, the individual byte code for each statement is displayed after each statement. For programs that are linked, the complete bytecode (including all statement label mapping) is displayed after the initial **`PROGRAM`** statement. *This is the same output as the **`DISASM`** command.*

In addition, any program that has used the **`DEFFN`** statement to define local functions will have the Byte-Codes for each function statement displayed after the program listing.

```
SHOW PROGRAM(PROFILE) MYPGM
```

This command outputs the same information as the **`SHOW PROGRAM(DISASM)`** option above, with the addition of counters for each bytecode showing how many times it has been executed. This is only displayed for linked programs; once a program is modified or unlinked this data is lost. This is used to determine what parts of a program are being most heavily executed to look for performance improvements or optimizer opportunities.

SHOW STATUS

The **SHOW STATUS** command displays a summary of information about the current session. Here is an example output:

```
JBasic Status as of Sat Sep 13 13:24:52 EDT 2008
   JBasic version:        2.6 (Built Fri Feb 20 23:21:37 EDT 2009)
   Current program:       PI_TEST
   Global symbol table:   53 entries
   ON Statement stack:    0 call levels
   Open files:            3 files
   Stored programs:       43 programs
     User programs:       26
     Functions:           14
     Verbs:               3
     Tests:               62
   Error messages:        177 messages
   Uptime:                12 minutes, 47 seconds
   Statement Statistics:
      Executed:           1261
      Compiled:           7625
      Exec as bytecode:   146
   Instructions executed: 146053
   Memory:
     In use:   4983728  Free:     2815056  Total:    7798784

   Options:              PROMPT="BASIC> ",
                         INPUTPROMPT="? ",
                         DEBUGPROMPT="DBG> ",
                         LANGUAGE="EN",
                         LABELWIDTH=10,
                         OPTIMIZE,
                         NOTRACE,
                         TOKENIZE,
                         NOSTATIC_TYPES,
                         NOPOOLING,
                         AUTOCOMMENT,
                         NOAUTORENUMBER,
                         TIMEGC,
                         PROMPTMODE,
                         NOSTATEMENT_TEXT
```

This information is a summary of data that can be collected from various global symbol table variables or runtime functions, but is presented as a summary of the status for the convenience of the user.

SHOW SYMBOLS

The **SHOW SYMBOLS** command is used to display the currently active symbol table, or specific symbol tables, or the entire tree of symbols. For each symbol, the name of the symbol, it's value, and additional descriptive information is displayed on the console. For example:

```
DBG> SHOW SYMBOLS

Table: Local to BAR
   $ARGS = arrayValue[0 values] = [ ] (readonly)
   $MODE = "CALL" (readonly)
   $PARENT = "FOO" (readonly)
   $START_TIME = 1.182352703804E12 (readonly)
   $THIS = "BAR" (readonly)
   DEPT = "Stock Management"
   PAY = 793.875

Table: Local to FOO
   $ARGS = arrayValue[0 values] = [ ] (readonly)
   $MODE = "RUN" (readonly)
   $PARENT = "FOO" (readonly)
   $START_TIME = 1.182352703804E12 (readonly)
   HOURS = 43.5
   MYNAME = "Tom"
   RATE = 18.25

Table: Local to Console
   $PARENT = "Console" (readonly)
```

This command displays the current symbol table, and all parent symbol tables up to and including the command shell symbol table. If the user ran program `FOO` that called program `BAR` and the **SHOW SYMBOLS** command was issued in program `BAR` (or from the debugger while running `BAR`) then the default symbol table display would be the symbols for `BAR`, the symbols for `FOO`, and any symbols created at the command shell level.

By default, hidden symbols are not displayed. A hidden symbol is one whose name starts with two underscore ("__") characters. These are symbols that may be created by JBasic to support execution of a statement or other operation. Additionally, symbols created by the debugger to track the current line and program are stored as hidden symbols. You can display all symbols including the hidden ones by using the **SHOW HIDDEN SYMBOLS** command.

The output of **SHOW SYMBOLS** includes that all programs have some read-only symbols created by JBasic that indicate the current program's name, how it was called, who called it, what arguments if any where passed to it, and when it started running.

In addition, the above example output shows variables created during execution of each program, such as `HOURS`, `RATE`, and `DEPT`.

You can place a table descriptor name after the **`SHOW SYMBOLS`** command to display a specific table only rather than the currently active symbol tree. These keywords have the following meanings:

Table Keyword	Description
`LOCAL`	Display only the symbol table for the current program.
`PARENT`	Display the symbol table for the caller of the current program.
`GLOBAL`	Display the system global symbol table
`MACRO`	Symbols used for macro substitution
`ALL`	Display all active symbol tables.

SHOW VERBS

The **SHOW VERBS** command displays information about all user-written verbs added to JBasic. A user-written verb is a JBasic program that uses the **VERB** keyword instead of the **PROGRAM** keyword to define its entry point. Verbs can be invoked just like JBasic statements, and the text of the statement following the verb is available to the program to control its operation. See the documentation on the **VERB** statement for more information.

Verbs can be defined as part of JBasic itself, or loaded from a workspace or a text program. The **SHOW VERBS** command displays the verbs that are available:

```
BASIC> show verbs

Verb programs:
 CLS                                14 statements      0 invocations
 DEAL                               11 statements      0 invocations
```

In this example, there are two verbs currently defined, **CLS** and **DEAL**. The information displayed about them indicates how many statements are in each program and how many times each program has been invoked as a verb.

You can display the program text of the verb by naming it on the command, as in this example:

```
BASIC> show verb deal

PROGRAM VERB$DEAL

  100           VERB DEAL
  110
  120           LET COUNT = 5
  130           IF LENGTH( $ARGS ) > 0 THEN COUNT = NUMBER( $ARGS[ 1 ] )
  140
  150           LET CARDS = RANDOMLIST( 1, 52 )
  160
  170           FOR I = 1 TO COUNT
  180             PRINT CARD( CARDS[ I ] )
  190           NEXT I
  200           RETURN
```

This shows the text of a verb (all verb's have a prefix of `VERB$` to differentiate them from program or function names). The program uses the `$ARGS` array that contains the keywords from the command line to decide how many cards to deal.

If the verb is an internal Java class such as **PRINT**, the **SHOW VERB** command displays that information.

SHOW VERSION

The **`SHOW VERSION`** command displays the current version of the JBasic executable jar file.

```
      SHOW VERSION
```

This is the same value that is stored in the variable `$VERSION` that is located in the `ROOT` symbol table. The string contains the program version and the date that the main entry point was last compiled.

TEST

The **`TEST`** command invokes built-in unit tests that are used to validate the correct behavior of JBasic. **`TEST`** is a built-in verb that scans the list of loaded programs for tests, which always begin with the string "`TEST$`".

Note that the **`TEST`** verb is loaded from an internal workspace that also contains all the actual test programs. Therefore, there will be no **`TEST`** programs visible until you issue the **`TEST`** command for the first time. After that, there will be additional programs in memory, all with the "`TEST$`" prefix.

```
TEST
```

You can get a list of the available tests by issuing the command with no arguments. If the tests are not in memory yet, this command will cause them to be loaded. After issuing this command, a **`SHOW TESTS`** command will list the tests by name.

```
TEST ALL
```

This causes all the tests to be run. The program reports any test that fails.

```
TEST ERROR1 ERROR2
```

This executes just tests `ERROR1` and `ERROR2`. You may specify as many tests as you wish on the command line. You can also put the keyword **`TRACE`** on the command line and it will cause the individual test programs to be traced, so each statement executed is displayed on the console - this is useful when debugging a failed test case, for example.

TIME

The **`TIME`** statement takes another statement and executes it, and then prints the number of statements executed (if the statement is a **`RUN`** or **`CALL`**) and the elapsed time.

```
TIME CALL BENCHMARK()
```

This invokes the statement (which itself calls a program in memory) and displays the results.

Because benchmarking often involves running the same test several times and averaging the results, this capability is built into the **`TIME`** command.

```
TIME(100) CALL BENCHMARK()
```

This executes the command 100 times and prints the average. When the count is greater than 1, the output includes the number of iterations and the total elapsed time. The count must be an integer constant; expressions are not permitted.

TRACE

The **TRACE** statement takes another statement and executes it, and prints the JBasic statements and byte code execute to perform the operation.

```
TRACE PRINT “Answer = “; X+3

ByteCode 00000: Stack element[  0]  - no object on stack!
         00000:     _STMT          0 "TRACE PRINT "Answer = ";X+3"

ByteCode 00001: Stack element[  0]  - no object on stack!
         00001:     _TRACE          1

ByteCode 00002: Stack element[  0]  - no object on stack!
         00002:     _STMT          0 "PRINT  "Answer = ";X+3"

ByteCode 00003: Stack element[  0]  - no object on stack!
         00003:     _STRING             "Answer = "

ByteCode 00004: Stack element[  1] string object(1089cc5e) = "Answer = "
         00004:     _LOADREF             "X"

ByteCode 00005: Stack element[  2] int object(46c837cd) = 55   <symbol>
         00005:     _ADDI          3

ByteCode 00006: Stack element[  2] int object(2c79809) = 58
         00006:     _CVT         21

ByteCode 00007: Stack element[  2] string object(2c79809) = "58"
         00007:     _CONCAT

ByteCode 00008: Stack element[  1] string object(7dce784b) = "Answer = 58"
         00008:     _OUTNL          0

Answer = 58
ByteCode 00009: Stack element[  0]  - no object on stack!
         00009:     _NOOP

ByteCode 00010: Stack element[  0]  - no object on stack!
         00010:     _TRACE          0
```

UNLINK

The **UNLINK** statement forces the removal of the linked ByteCode, and the recompilation of the individual statements. This is the same operation that is performed when a new statement is added to the program or a statement is deleted.

```
    UNLINK
```

When a program is first loaded from the workspace, it is compiled but not linked. That is, each statement is compiled as it is entered, but the byte-code is stored with each statement object.

When the link is performed, each statement's byte-code is collected up and stored in the program executable byte-code object. This is the byte-code that is actually run when a program is executed.

When a new statement is added or deleted from the program, the linked byte-code is destroyed. This is because new statements can change the relationship between statements, add labels, or delete code.

When a program is run, if it has not been linked, it is linked automatically before execution (this includes functions and subprograms). However, you can use the **LINK** command to force the link operation. This is most useful in combination with the **DISASM** command to view the completed program or to review the effectiveness of the optimizer.

To return the program to the pre-optimized state, use the **UNLINK** command to delete the linked program and recompile the individual statements.

Product Revision History

This section summarizes the product revision history. Detailed descriptions of most changes made in the last several years can be found in the files `CHANGES-ARCHIVE.TXT` and in the `CHANGES` section of the help file `JBASIC-HELP.TXT`, both of which are found in the root level of the Eclipse project directory for JBasic.

Version	Date	Description
1.0	2002	JBasic began as a learning experiment in programming Java, and started as nothing more than an expression evaluator.
1.1	2003	Almost a year after writing the expression evaluator, I wanted to try to learn some new Java again and added some elementary language elements to JBasic. This included the idea of programs, branch and subroutine functions, and a move towards a BASIC-like dialect.
1.2	Aug. 2004	Early 2004 saw the introduction of the first program IO, the Library as default location to load programs from, and user-written verbs. HELP was the first user-written verb. Structured symbol tables and symbol attributes added.
1.3	Sept. 2005	ANT-based builds, user documentation, and allowing JBasic to be used within another Java program. Signals and messages added. Lots of small bug fixing and feature additions like new functions, etc.
1.4	Jan 1, 2006	Java programs can add statements to the language. ByteCode introduced to convert from a lexical interpreter to a ByteCode compiler. ByteCode linker added to create a single stream of ByteCodes to execute a program. DATA statements added to language. JavaDoc cleanup means that the generated documentation can be used by a Java programmer to include JBasic in their program.
1.5	Mar 1, 2006	Repackaging into sub-packages as JBasic becomes more than 100 classes. ByteCode execution moved from large "switch statement" to individual classes for each opcode. Binary file IO added. Major bug fix push in improving language features. ByteCode optimizer added to compile and link phases. JBasic object data types introduced.
1.6	Sept. 2006	Debugger added to assist JBasic program development. JDBC support added with DATABASE file type. Thread-safe changes created to allow user THREADS, with QUEUE pipes to communicate between threads. Major performance improvements and bug fixing. GW-BASIC compatibility, full line number support (including RENUMBER) added.
1.7	Jan. 2007	Improved "BASIC-like" program LOAD and SAVE operations in addition to the WORKSPACE model. New input and output format features. Another major round of bug-fixes. Added COMMON blocks to share variables between CHAIN programs. New features for data type declaration, strong typing of variables.
2.0	Nov. 2007	Added MULTIUSER mode based on threading and program/symbol isolation work. Added the open source TelnetD implementation by Dieter Wimberger to support this server mode.
2.1	Jan. 2008	Bug fixes, with some internal Java code cleanup and updated documentation.

Version	Date	Description
2.2	May 1, 2008	More bug fixes. Internal error handling model for runtime mode changed to be based on Java exceptions which encapsulated Status objects. Cleanup of online help and documentation for spelling and grammar goofs.
2.3	Jun 28, 2008	Added XML support for storing or transmitting JBasic values using standards-based representation. Added SAVE XML to save a program as XML, which can be reloaded using the LOAD command which automatically detects XML. You can input an XML string as a value by using INPUT.. AS XML. This release also has major bug fixes for file path handling in multi-user mode, so that logical names are handled consistently and physical paths are not visible to remote users.
2.4	Nov 30, 2008	Primarily a bug-fix release, but added complex statement notation on the left side of an assignment operation and added PRINT X= notation that prints the variable name as well as the value. Surfaced COMPILE() which compiles a statement into a bytecode variable, and added ASM USING which uses that array to process the bytecodes for execution. Added INPUT BY NAME for user-directed variable input. Added ON..THEN GOSUB to provide subroutine support for error handling. DO WHILE..LOOP and DO UNTIL..LOOP syntax was added to allow loop control to occur at the top of a loop rather than the bottom.
2.5	Feb 7, 2009	Improved support for automatic validation and creation of user directories in SERVER mode, and default logical name creation. Added new PIPE file type. Added FLOAT, VARYING STRING, and UNICODE STRING binary file record types. Added sizes for INTEGER and FLOAT binary types. Added a FIELD statement to replace the RECORD statement, which also allows binding of a record definition to a file for compatibility with other variants of BASIC. Added multi-line IF statements. SUB statement allows local subroutines and functions. Added the "Compatibility" section of this manual.
2.6	May 2009	Added new support for direct manipulation of Java objects from within JBasic. Java objects can be created in JBasic or created outside JBasic and passed in for use by JBasic. Additionally, object methods and fields are directly accessible to JBasic programs. Created new preferred syntax for DIM statements that mirrors the FIELD statement syntax with the type before the variable name. Programs can be called as functions if they have a RETURN statement. Functions can explicitly set the type of the return value in the FUNCTION declaration. Documented LOCK, UNLOCK, and related commands that surface blocking locks available to JBasic threads to protect critical regions.

Version	Date	Description
2.7	October 2009	Lots of bug fixes dominate this release. This includes internal simplifications for classes such as Value, and removal of vestigial code left over from when programs didn't have to be linked to run. The Java stack trace mechanism is partially replaced to allow much faster throws of signals. Status classes can be more usefully nested, and reporting of such errors is now more human-readable. Significant cleanup and improvement was also done across the board in statement compilers, so poorly constructed statements were more accurately detected and reported. Now, statements in error are left in the program, but attempts to `RUN` or `LINK` the programs reports the compile errors as unresolved. Numerous byte code classes where corrected, particularly with respect to when a Value reference versus a copy of the Value reference were required.
2.8	Dec 2010	Introduced TABLE data type which is used to store rectangular regular arrays of data similar to a database table. JOIN, WHERE, and SORT operators are added to allow construction of queries. Added socket file types so a client/server set of programs can be written. Added URL parsing operations and indirect file identifiers so a single server can handle multiple client connections. Introduced pre- and post-increment and decrement operators. Added arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE. SAVE PROTECTED now uses encrypted XML to store protected code. Added selector and indicator arrays as a method for selecting subsets of an array. Added simple macro facility for creating program templates.
2.9	TBD	Update the SYSTEM command to allow spaces, etc. in command arguments. Added pattern-matching optimizer (PMO) that is extensible using XML. Moved error messages and permission descriptions to external localizable XML. Added SQL processing statements like CREATE TABLE, INSERT, and SELECT. Added SQL() function to dynamically execute commands. Allow SELECT to operate on JDBC databases with pass-through of WHERE clauses.