HLABasic

> **Warning**: This program was written for fun. Work on it was suspended when other projects started entering "crisis mode." There are still a few features missing and the code has not been tested. Use this code at your own risk. The source listings are available, so feel free to fix any problems you find.

HLABasic is a traditional BASIC interpreter. The prefix "HLA" refers to the fact that this interpreter was written with the High Level Assembler (and, therefore, the source code to the interpreter is in HLA). HLABasic does not particularly possess any "higher level" features because of this name prefix. The language itself is reminiscent of the microcomputer BASICs that appeared in the early 1980s. This document briefly describes the language features of HLABasic for those who are interested in writing HLABasic programs; this document also describes the internal organization of HLABasic for those who are interested in extending or modifying HLABasic.

Although HLABasic is a relatively full implementation of the BASIC language, one should not try to compare this language against behemouths like Visual BASIC. HLABasic does not contain "rapid application development" or GUI features in the language. It's strictly a text-based development and execution environment. The IDE (if you're willing to call it that) is a simple line-oriented text editor reminiscent of the early BASIC interpreters. This is not what most people would consider a modern BASIC-based software development system. However, you do have the sources (and the code is all public domain) so you can easily extend the language and software development system in any way you see fit.

The real purpose of HLABasic is to provide an example of a decent, though plain, BASIC interpreter that programmers can incorporate into other projects; e.g., HLABasic makes a great scripting language. Further, it provides a decent implementation of the BASIC language for those how are simply interested in "seeing how it's done." Again, the code is public domain, so there are absolutely no restrictions on what you can do with this program.

HLABasic does have a few current limitations. First of all, it doesn't support file I/O. This is easy to fix and will probably be added to a future version of HLABasic. Second, HLABasic doesn't support user-defined functions. Single-line functions are possible (and not that difficult to add), but multi-line functions would be difficult to add because the internal code is not re-entrant (a general requirement for multi-line functions). Formatted output in HLABasic is certainly possible, though it uses a different syntax than traditional BASIC interpreters. Finally, do keep in mind that HLABasic is interpreted rather than compiled, so the performance is going to be quite a bit lower than an equivalent, compiled, program (though do note that HLABasic is a relatively high performance interpreter, handily beating many other interpreters out there).

HLABasic does support a few extended features of its own. One of HLABasic's big improvements over the tranditional BASIC language is that all variables are automatically typed. HLA supports integer, real, and string objects and HLA automatically converts between these types as needed (and if possible). HLA supports dynamically allocated arrays whose size can change during program execution. Arrays may contain any reasonable number of dimensions and, in fact, each row of multi-dimensional array can have a different size. Although HLABasic uses line numbers for text editing purposes, it does not use line numbers as statement labels. Instead, you can use generic identifiers as the target of GOTO and other control transfer instructions.

HLABasic provides powerful string manipulation functions as well as several built-in mathematical functions. For those familiar with the HLA assembler, it's quite easy to extend the set of built-in functions by modifying the HLABasic source code.

Although by no means "tiny", the HLABasic interpreter does not consume tremendous amounts of system resources. The BASIC interpreter itself, along with all the static variables, consumes about 64K of memory (this does not include space for user programs or data, which is obviously application dependent). Although it wouldn't be that hard to shrink the code to one-half this size, saving 32KBytes on modern machines (that have 128MBytes or more of memory) doesn't seem like a very cost-effective use of programmer time. Further, one of the reasons the code is a tiny bit larger than it needs to be is because the interpreter was organized in order to allow easy modification. Shrinking the code would reduce the generality and make it more difficult to extend.

The HLABasic interpreter is a high-performance interpreter. The development environment "tokenizes" all input programs and stores them in a compact and easy to interpret format in memory. This saves considerable

time during run-time because the interpreter can quickly figure out how to execute a statement without first running some pattern matching algorithm on the statement in order to figure out what statement it is. Furthermore, immediately upon receiving the "RUN" command, the HLABasic interpreter makes a quick scan over the (tokenized) source code and patches in target addresses so that the interpreter doesn't have to search for a label each time it encounters a control transfer statement during execution. This provides a dramatic improvement in performance, especially for larger programs.

There are a few areas where the performance of HLABasic could be improved. First of all, the expression evaluator uses a "recursive-descent" parsing technique that isn't particularly high-performance. One could easily swap in a bottom-up operator-precedence parser for this code and see an immediate performance boost. Better yet, the tokenizer could be rewritten to emit tokens in "reverse-polish notation" and that could boost the performance considerably. This latter technique will probably appear in a later version of HLABasic.

Although HLABasic is written in assembly language, the code was designed to be easy to read and modify, not to be the fastest executing code possible. A good assembly language programmer will find lots of opportunities to speed up the interpreter by scheduling instructions and choosing better instruction sequences that appear in the code. Again, HLABasic's source code was designed to be easier to read and modify rather than worrying about extracting the last cycle from the execution time of the program. HLABasic gets its performance from the choice of good data structures and algorithms, not via fancy assembly code. This leaves lots of opportunity for those who like to speed up assembly code. One should easily be able to double the speed of the interpreter just through the use of proper instruction scheduling. That, however, is left as an exercise to the reader (no plans exist to incorporate such improvements in future versions because the code needs to remain maintainable and readable).

> **Warning:** *HLABasic is not a "commercial-quality" piece of software. It has had a small amount of testing, but it certainly contains defects that will cause problems to those who use HLABasic for serious software projects. The author(s) make no warrantees about the qualify of HLABasic (other than a lack thereof) and cannot take responsibility for the use, misuse, or abuse of this code. The end user of this program is solely responsible for ascertaining the fitness of this software product and correcting any latent defects appearing therein.*

Well, without further ado...

1.1 The HLABasic Development Environment

For those unfamiliar with the original BASIC program development environment (which HLABasic uses), HLABasic is going to come as a bit of a shock. The editor is a simple line-based editor and the development environment only supports a hand-full of commands that you enter from the keyboard in a "command window" fashion. HLABasic's development environment doesn't support the mouse, fancy graphics, or a GUI. It's strictly a text-based system. On the other hand, its very simplicity makes it easy to learn.

Running HLABasic is very easy. Either double-click on the HLABasic.exe icon from Windows or type "HLABasic" from a command line prompt. HLABasic is a command-window / console-mode program. So if you run it directly from Microsoft Windows, it will open up a console window and wait for your command. After running HLABasic, the program will display a short sign-on message and present you with a ">" prompt character. At this point, HLABasic is ready to accept a command.

An HLABasic command begins with either an integer in the range 1..65535 (which denotes a BASIC statement) or one of the following reserved words:

```
dump, list, edit, run, new, debug, load, save, or quit
```

Later subsections describe each of these commands and their syntax in detail.

If an HLABasic command begins with an integer value in the range 1-65535, then the HLABasic interpreter processes this command as an HLABasic program statement. Like the original BASIC language definition, HLABasic attaches a line number to each statement in the program. Program source lines are stored in memory sorted by their line numbers. So if you enter three source lines into an HLABasic program with line numbers 325, 623, and 124, then HLABasic treats the line with line number 124 as the first line of source code, the state-

ment with line 325 as the second line of source code, and the line with number 623 as the third line of source code; this is true regardless of the order you enter the lines into the source file.

Generally, you do not want to use sequential line numbers for adjacent lines in the source file. A BASIC tradition is to number sequential lines in the source file using line numbers that increment by 10. This allows you to easily insert additional statements between two statements in the source file at a later date. Currently, HLABasic does not support a "Renumber" command[1], so if you number your statements sequentially with no line number gaps, HLABasic will not allow you to insert new statements between such lines. Therefore, it's a good idea to leave at least 10 "slots" open between each statement (more if you're not sure how much modification your program will sustain over its lifetime). Note that, unlike the original definition of BASIC, HLABasic does not use line numbers as targets for GOTO and other control transfer statements; HLABasic only uses line numbers to place lines in the source file.

If the line number you specify does not exist in the source file, then HLABasic insert the specified BASIC statement into the source file. If the line number already exists in the source file, HLABasic replaces the existing line of text with the new line. To delete a line of text from the source file, simply specify a line number as a comment by itself (i.e., no source text).

To save space in memory and to improve the run-time performance of your programs, HLABasic "tokenizes" input statements. This means that HLABasic replaces strings like "PRINT" or "GOSUB" with a single byte that represents the BASIC keyword. HLABasic also removes all whitespace outside of string constants for this same reason. Therefore, whenever you list an HLABasic program, the output may not be exactly the same as the lines of text you input. HLABasic's LIST command strives to produce readable output, but it may not produce the same characters you provided. In particular, HLABasic strips away all excess whitespace between items on a source line; hence, if you insert extra space in the middle of an HLABasic statement, HLA will remove those spaces when you list the file. Note that HLABasic preserves the spaces appearing between the line number and the first keyword on the line. This maintains indentation on the line[2].

For similar efficiency reasons, HLABasic also converts integers to their four-byte representation and then converts this integer back to a string when listing the program (this is done for performance, rather than space, reasons since most integer constants in a typical BASIC program are less than 1,000, so they would normally require fewer than four bytes to represent; this storage scheme, however, saves the expense of translate the text to the integer format each time the interpreter encounters the constant during execution). One side effect of this scheme is that HLABasic will eliminate any leading zeros you attach to an integer constant; i.e., if you input a source line containing the (legal) constant 0012340, HLABasic will display 12340 when you list the source file.

For run-time efficiency reasons, HLABasic also converts floating point constants to their eight-byte internal format and stores this eight-byte value as part of the tokenized code. However, HLABasic also preserves the character string representation of that floating point value. This is done to improve run-time performance (converting a string to a floating point value is a very costly operation). HLABasic also keeps the text form around because floating point values are approximations; converting the internal format back to a string may not produce exactly the same string as originally supplied. Furthermore, it is very difficult, when given the internal representation for a number like 1.2345e6 to determine whether the user originally input 1.2345e6, 1234500, 123.45e4, etc. Therefore, HLABasic stores the string format along with the binary format so that it can print exactly what the programmer specified when entering the source code in the first place. This almost always guarantees that the tokenized form takes more memory than the string form, but the run-time performance benefits make this a good trade-off (memory vs. speed).

Most of the time, HLABasic tokenized code is smaller than the equivalent text form of a statement. However, as you've seen above, there are some cases where certain tokens actually consume more space than their text representation. On the average, however, tokenized statements are shorter than the pure text form. If you're interested in learning more about HLABasic's tokenization, please consult the HLABasic source code.

Each of the following sections describe the available HLABasic commands. Each command begins with a reserved word with some possible parameters. See the sections below for complete details.

---

1. This is an easy feature to add to HLABasic, it just wasn't present in HLABasic v1.0. It will probably appear in a later version.

2. HLA doesn't actually preserve the spaces, it simply counts them and then stores a single eight-bit count specifying the number of spaces preceding the first keyword. When you list the source file, HLABasic automatically expands this byte to the specified number of spaces.

1.1.1 Dump

The DUMP command is intended for internal development use only (i.e., for those working on the HLA source code). It displays the "byte codes" for a given sequence of statements in an HLABasic program. This allows a developer to verify the correct tokenization of HLABasic statements in the program. There are three forms of this statement:

```
dump
```

This variant dumps out the entire program in a hex format.

```
dump #
```

This variant dumps only the specified line number in a hex format.

```
dump #1, #2
```

This variant dumps out the data for all the source lines between line number #1 and line number #2 in a hexadecimal format.

In addition to dumping out the tokenized data, the DUMP command also displays a list of variables defined in the program as well as memory starting and ending addresses for the source file.

About the only interest an HLABasic programmer would have in this command is that it lets them see how much memory a particular statement consumes, so they could use it to determine if one form of a statement is more efficient than another.

1.1.2 List

The LIST command displays the lines of source code in an HLABasic program. This command takes the following forms:

```
list
```

The LIST command, by itself, lists the entire program.

```
list #
```

With a single line number as a parameter, the LIST command displays the specified line.

```
list #, #
```

With two line numbers, separated by a comma, as the LIST parameters, the LIST command displays all the lines in the source file between the first and second line numbers.

Remember, LIST may not produce an exact representation of the source program as it was input. In particular, certain spaces and leading zeros (on integers) are removed. There may be other minor changes on output as well. Semantically and syntactically, however, the statements that LIST produces are identical to the original user input. Note that HLABasic may insert extra spaces into the middle of a statement. For example, if you have an expression like "x*y" the LIST command may display this as "x * y" in order to make the code more readable. This means that certain statements may actually take up more space on the output line than appeared originally on input.

1.1.3 Edit

The EDIT command provides a convenient way to edit existing lines of text in the source file. The syntax for this command is

```
edit #
```

This command displays the specified line number and inserts the line into the system input line buffer allowing you to edit it with the arrow keys (just as you would a command window statement). You may use the Insert key to toggle between insert/replace mode and you can use the Delete key to delete characters from the line.

1.1.4 Run

This command begins the execution of the HLABasic program in memory. This command has no parameters, so the syntax is simply:

```
run
```

When you execute the RUN command, HLABasic first makes a quick "compile" pass over the source file to fill in addresses of target labels, variables, and other values that improve the run-time performance. On most modern machines, this pass over the source file is so quick that you won't even notice it.

1.1.5 New

The NEW command erases the program in memory in order to allow you to enter a new BASIC program into the workspace. This command has no parameters, so the syntax is simply:

```
new
```

Warning: using the New command irretrievably deletes the program currently in memory. You should save any important work before using this command. There is no "undo" available here.

1.1.6 Debug

The DEBUG command activates the statement trace facility in HLABasic. This command has no parameters, the syntax is simply:

```
debug
```

By default, the trace option is turned off in HLABasic. Executing the DEBUG command toggles the value between on and off. Whenever you type this command at the HLABasic prompt, the HLABasic system will respond with "debug on." or "debug off." that tells you the new state of the trace mode.

While the trace mode is on, HLABasic will display the line number of each statement it executes while the program is running. This allows you to trace through a program and help you determine where the program fails. Of course, executing a program with the trace turned on completely destroys any on-screen formatting your program produces, so you should only turn this on when you need to actually trace through some code.

1.1.7 Load

The LOAD command loads an HLABasic source file from the disk. The syntax for this command is

```
load filename
```

The filename parameter must be a valid Windows filename. HLABasic does not allow spaces within a filename. The conventional suffix for an HLABasic program is ".hb". Example:

```
load HelloWorld.hb
```

Note that HLABasic saves programs on the disk in a text, not tokenized, format. Therefore, you may edit HLABasic programs you save to disk with any text editor. Note, however, that HLABasic assumes that all disk-based files are syntactically correct, so if you make a typo when editing the program with an external text editor, HLABasic may not load the file properly into memory (the load command effectively reads the text from the specified source file exactly as though you were typing the data at the keyboard; if there is a syntax error, then HLABasic prints the offending statement(s) and an appropriate error message. The statements are not stored in the source file.

Also note that the LOAD command first erases any program currently in memory. Therefore, you should always save your important programs in memory before using the LOAD command.

1.1.8 Save

The SAVE command saves the HLABasic source file in memory to a disk file. The syntax for this command is

`save` *`filename`*

This command writes the source file as text to the specified file. You may edit this file with any editor that is capable of working with standard text files (note, however, the warning in the section on the LOAD command). If the file already exists, HLABasic will overwrite that file's data with the HLABasic program currently in memory.

By convention, you should attach a ".hb" suffix to HLABasic source files you save on the disk.

1.1.9 Quit

This command, which doesn't allow any parameters, quits HLABasic and returns control to Windows. Syntax:

`quit`

Note: pressing control-C also does the same thing as the QUIT command.

Warning: this command doesn't save the program currently in memory. Be sure to save your data before quitting HLABasic if that data is important.

1.1.10 Commands That Would be Useful to Add to HLABasic

There are a few commands that would be easy to add to HLABasic but haven't yet been added (due to a lack of time). This section describes a few such commands:

`renumber #`

Renumbers all the lines in the BASIC program starting with line number "#" and adding "#" to each sequential line thereafter.

`renumber #1, #2, #3`

Renumbers statements #1 through #2 using #3 as the sequence value to add to each line number. (Issue: what happens if line numbers overlap after doing this?)

`delete #1, #2`

Deletes lines #1 through #2 in the source file.

Obviously, there are several enhancements that the existing commands could use (e.g., warning people when they are about to lose data). Such "Gold Plating" was left out of HLABasic due to time constraints plus the desire to keep the code relatively small.

1.2 HLABasic Statements

An HLA source line takes the following general format (braces surround optional items).

```
line_number  {optional_label: }
                  statement #1 { : statement #2 { ... { :statement #n}...}}
```

That is to say, a single source line in HLABasic may contain multiple statements separated by colons. A single statement label, consisting of an identifier immediately following by a colon, is legal at the beginning of the line. A label may appearon a line by itself. There are some exceptions to this general syntax, however. A few statements must appear at the beginning of a source line and a few statements must be the only statement on the source line. We'll discuss these exceptions as we encounter them. Although there is a very slight performance benefit to placing several statements on a single line, good programming style generally suggests that you place

one statement per source line in an HLABasic program. Placing multiple statements per line should be reserved for those times when you underestimated the number of statements you needed to insert between two existing source lines when choosing line numbers (and that reason would quickly go away with the creation of a RENUMBER command). Therefore, the rest of this document will assume one HLABasic statement per source line unless there is good reason to violate this.

HLABasic limits source input lines to about 128 characters per source line (actually, this is a Windows console window limitation more than an HLABasic limitation). Because it is possible for HLABasic to expand your source lines when listing them, you should keep your lines less than about 80 characters or so in order to allow for this expansion. Don't forget that the SAVE command is effectively a "LIST to file" command. If you cram as many characters per line as you can and then save the file to disk, HLABasic may reject the input lines if they are too long (because of expansion).

1.2.1 Expressions and the LET / Assignment Statement

All HLABasic statements except one must begin with an appropriate keyword. The single exception is the LET statement. The LET keyword is optional for this statement. This is the only statement that does not require a keyword to lead off the statement. Note that when you list the statement, HLABasic inserts the LET keyword. This statement uses the following syntax

```
{LET} variable = expression
```

"*variable*" denotes an HLABasic identifier and "*expression*" denotes a valid HLABasic expression.

Variable names and other identifiers in HLABasic must begin with an alphabetic character. You may follow this first alphabetic character with any number of alphanumeric and underscore characters. There is no limitation on identifier name lengths other than the practical limit of the source line length. Due to a design flaw in HLABasic, identifiers are currently case sensitive. Therefore, "i" and "I" are two separate identifiers in the HLABasic language. Do not count on this feature, however, as this flaw will be corrected in a future version of HLABasic.

For the LET statement, the variable identifier must be an as-yet undefined symbol or it must be a symbol that appears as an identifier in another statement. In particular, it cannot be the same identifier as some statement label in the program.

Note to long-time BASIC users: you do not attach suffixes like "$", "!", or "%" to an identifier to denote the type of that symbol. HLABasic automatically keeps track of the variable's type and dynamically changes the type as necessary during program execution. This is a very powerful feature of the HLABasic language that simplifies many programming tasks. If the value of *expression* is a string object, then HLABasic dynamically converts the variable to a string variable; ditto for integer and real expressions (HLABasic converts the *variable* to an integer or floating point object, respectively).

1.2.1.1 Operators in HLABasic

An expression in HLABasic consists of various *terms* and *operators*. The context free grammar for HLABasic expressions is the following[3]:

X → X andop L | L

L → L relop A | A

A → A addop M | M

M → M mulop T | T

T → variable | intconst | strconst | fltconst | functionID(X) | - T | NOT T

andop → AND | OR

3. If you don't understand context-free grammars, just realize that HLABasic's expression syntax is identical to other common high level languages, especially BASIC.

relop → < | <= | >= | > | <> | =

addop → + | -

mulop → * | / | %

The precedence of the operators is as follows:

Table 1 HLABasic Operator Precedence

Operator	Precedence	Associativity
- (unary), NOT	5	right-to-left
*, /, % (mod)	4	left-to-right
+, -	3	left-to-right
<, <=, =, <>, >, >=	2	left-to-right
AND, OR	1	left-to-right

You may use parentheses to override the default precedence. The relational and boolean operators produce an integer zero/one result denoting false/true (respectively).

HLABasic uses a *dynamic type system*. This means that HLABasic determines the type of an expression while it is calculating that expression, based on the type of the operands in that expression. HLABasic does not require you to declare the type of a variable before assigning some value to that variable; indeed, as the execution of a program progress, you can assign values of differing types to the same variable (this is similar to the *variant* type in many languages). HLABasic supports three primitive data types: integers, reals, and strings. HLABasic attempts to accomodate each of these types in an expression as best it can.

Obviously, most of the operators above are intended for use with numeric (integer and real) data types. Indeed, only the "+" (concatenation) and relational ("<", "<=", etc.) operators directly accept string operands. However, as you'll see in the next section, HLABasic actually allows string operands anywhere a numeric operand is legal (and vice versa). HLABasic will attempt to convert such strings into numeric values should they appear in such a location. The following table describes each of the above operators:

Table 2

Operator	integer OP integer (or OP integer)	integer OP real or real OP integer	real OP real (or OP real)	string OP numeric numeric OP string string OP string (or OP string)
- (unary)	Negates the integer value	N/A	Negates the real value.	Attempts to convert the string to an integer or real and then negates that value.
NOT	Sets the integer to zero if it was non-zero, one if it was zero.		Generates a run-time error.	Generates a run-time error.

Table 2

Operator	integer OP integer (or OP integer)	integer OP real or real OP integer	real OP real (or OP real)	string OP numeric numeric OP string string OP string (or OP string)
*	Computes the integer product of the two operands	Converts the integer to a real and computes the real product of the two operands.	Computes the real product of the two operands.	If both operands are strings, this generates an error. If one operand is numeric, HLABasic attempts to conver the other operand to a number and computes their product.
/	Computes the integer quotient of the two operands.	Converts the integer to a real and computes the real quotient of the two operands.	Computes the real quotient of the two operands.	If both operands are strings, this generates an error. If one operand is numeric, HLABasic attempts to conver the other operand to a number and computes their quotient.
%	Computes the integer remainder of the two operands.	Generates a type mismatch error.	Generates a type mismatch error.	If either operand is an integer, this expression attempts to convert the string to an integer and then computes the remainder of the two operands. If either operand is not integer, then this expression produces a type mismatch error.
+	Computes the sum of the two integer operands.	Converts the integer to a real and computes the real sum of the two operands.	Computes the real sum of the two operands.	If both operands are strings, this expression computes the concatenation of the two string. Else it attempts to convert the string to the same type as the other numeric operand and computes the sum of the values.

Table 2

Operator	integer OP integer (or OP integer)	integer OP real or real OP integer	real OP real (or OP real)	string OP numeric numeric OP string string OP string (or OP string)
-	Computes the difference of the two integer operands.	Converts the integer to a real and computes the real difference of the two operands.	Computes the real difference of the two operands.	If both operands are strings, this operator generates a type mismatch error. Otherwise, it attempts to convert the string to the same type as the other numeric operand and computes the difference of the values.
<, <=, =, <>, >, >=	Compares the two integer operands and returns 0/1 based on the comparison.	Converts the integer to a real and compares the two operands, returning 0/1 based on the result of the comparison	Compares the two real operands and returns (integer) 0/1 based on the comparison.	If both operands are strings, this operator compares the two strings lexicographically and returns 0/1 based on the result. Otherwise, it attempts to convert the string to the same type as the other numeric operand and compares the values.
AND	Returns one if both operands are non-zero, zero otherwise.	Causes a run-time error (type mismatch).	Causes a run-time error (type mismatch).	One one operand is an integer operand, this operator attempts to convert the string operand to an integer and then computes their logical AND. Otherwise, this operator generates a run-time error.
OR	Returns one if either operand is non-zero, zero otherwise.	Causes a run-time error (type mismatch)	Causes a run-time error (type mismatch)	One one operand is an integer operand, this operator attempts to convert the string operand to an integer and then computes their logical OR. Otherwise, this operator generates a run-time error.

1.2.1.2 Dynamic Typing

As noted earlier, HLABasic automatically sets the type of an expression based upon the types of the operands within that expression. Like most high level languages, HLABasic expressions produce an integer result if both operands are integers. For operators that support real operands (not all of them do), HLABasic produces a real result whenever either or both operands are real. This is a fairly intuitive process for those who are familiar with other high level languages. What is probably not intuitive is the behavior of strings within arithmetic expressions. The purpose of this section is to discuss the behavior of HLABasic's type system and the automatic conversions it provides.

Perhaps it is wisest to first consider those conversions that HLABasic will *not* do. In HLABasic, certain operators are defined to support integer operands only. This includes the mod/remainder operator ("%") as well as the logical operators AND, OR, and NOT. These operators do not allow real operands, even if that real value doesn't contain a fractional component (e.g., 1.0). If you really want to use real expressions with the AND, OR, and NOT operators, please see the INT and ROUND built-in functions that will convert a real value to an integer.

The NOT operator accepts only an integer operand. It will report a type mismatch error if you supply a string or real operand to this operator. This is true even if the string holds the character representation of some integer value. You must explicitly use the VAL function to convert the string to a value if you have an integer string. As noted above, you must use INT or ROUND to convert real values to integers before operating on them with the NOT operator. HLABasic does not support automatic type conversion of NOT's operands because it is very unusual to have non-integer operands after the NOT operator (i.e., this usually indicates some sort of error). Hence, HLABasic reports the error rather than attempting to convert the data to an integer format (which would be dubious, at best).

The AND and OR operators also require integer operands. However, if one operand is an integer and the other is a string, HLABasic will first attempt to convert that string to an integer result. If the conversion is not possible, then HLABasic reports an error. HLABasic does not allow either (or both) operands to be real values.

The mod/remainder operator ("%") requires integer operands. Like the AND and OR operators, this operator rejects any real operands. If one operand is an integer and the other is a string, this operator will attempt to convert the string to an integer before rejecting the expression.

The unary negation operator ("-") requires a numeric operand. If you supply a string value as an operand, HLABasic will not attempt to convert this to a numeric value; it will simply report a type mismatch error.

Most other operators support a variety of mixed-mode operators and will automatically convert their operands so that they have the same type. If the two operands are integer and real, HLABasic converts the integer to a real operand and then calculates the result based upon the real operands. If one operand is a string operand, HLABasic first attempts to convert the string to an integer value. If the string does not contain the string representation of a legal integer value, then HLABasic attempts to convert that string to a real value. If the string does not contain the legal representation of a real value, the HLABasic stops with a type mismatch error. If HLABasic successfully converts the string to an integer or real value, it completes the calculation as though the program had originally supplied that integer or real operand. Note that HLABasic will not attempt to convert string operands to numeric form if both operands around a numeric operator are strings.

1.2.1.3 Built-in Functions

HLABasic supports a large number of built-in functions. Furthermore, it is easy to add new functions to HLABasic since you have the source listings to the interpreter. This document will not discuss how to do that, but if there's a function you want in HLABasic that 's not currently present, keep this option in mind. The following subsections describe the default set of built-in functions.

In the following sections, the parameter and return types are denoted by using variable names with the following prefixes: "i" for integer operands, "f" for floating point/real values, "s" for string values, "n" for numeric (integer or real), and "a" for an arbitrary type. Note that HLA will automatically convert integers to real in parameter lists. Likewise, most function that expect a numeric value will accept a string and attempt to convert

that string to a numeric value. The following sections, however, describe the native types for the parameters (as well as the function return types).

1.2.1.3.1 acos

```
10 f1 = acos( f2 )
```

This function computes the arc-cosine of its parameter. The parameter must be a value in the range -1..+1. This function returns the angle, in radians, whose cosine produces the parameter value.

1.2.1.3.2 asc

```
10 i = asc( s )
```

This function takes a string parameter. It returns the ASCII code of the first character in the string. This function generates a run-time error if the string is empty.

1.2.1.3.3 asin

```
10 f1 = asin( f2 )
```

This function computes the arc-sine of its parameter. The parameter must be a value in the range -1..+1. This function returns the angle, in radians, whose sine produces the parameter value.

1.2.1.3.4 atan

```
10 f1 = atan( f2 )
```

This function computes the arc-tangent of its parameter. This function returns the angle, in radians, whose tangent produces the parameter value.

1.2.1.3.5 chr

```
10 s = chr( i )
```

This function produces a string of length one whose single character has the ASCII code specified by the integer parameter.

1.2.1.3.6 cos

```
10 f1 = cos( f2 )
```

This function computes the cosine of its parameter. The parameter must be an angle in radians. This function is most accurate when the angle is in the range -2*pi..2*pi radians.

1.2.1.3.7 exp

```
10 f1 = exp( f2 )
```

This function computes e^{f2}.

1.2.1.3.8 int

```
10 i = int( f )
```

This function extracts the integer portion of a real value and converts it to an integer (via truncation). The resulting value must fit in a 32-bit integer.

1.2.1.3.9 left

```
10 s1 = left( s2, i )
```

This function extracts the left-most "i" characters from the string "s2" and returns this substring.

1.2.1.3.10 len

```
10 i = len( s )
```

This function returns the current length of the string variable "s" as an integer value. Note that there is no practical length on strings in HLABasic.

1.2.1.3.11 log

```
10 f1 = log( f2 )
```

This function computes the natural logarithm (base-e) of f2. The value of the parameter, f2, must be greater than zero.

1.2.1.3.12 mid

```
10 s1 = mid( s2, i1, i2 )
```

This function extracts a substring of length i2, starting at character position i1, from s2 and returns the result. Note that string indexes are zero-based in HLABasic. Hence, MID(s2, 0, i2) is equivalent to LEFT(s2, i2).

1.2.1.3.13 randomize

```
10 i = randomize
```

This function "randomizes" the HLABasic random number generator and then returns a random number. Note that this instruction uses the x86 RDTSC instruction which is only available on Pentium and later chips.

1.2.1.3.14 right

```
10 s1 = right( s2, i )
```

This function extracts the right-most i characters from string s2 and returns the result.

1.2.1.3.15 rnd

```
10 i = rnd
```

The RND function returns a pseudo-random number on each successive call. Note that, in the absence of calling RANDOMIZE, the RND always produces the same sequence of pseudo-random numbers each time you start the HLABasic interpreter (but not each time you run a BASIC program from within HLABasic without restarting HLABasic). Therefore, if you need to guarantee the same sequence on each run (i.e., during testing) then you should restart HLABasic before each run.

To get a more random sequence, you should call the randomize function (just once) at some point in your program. The RANDOMIZE function reads the Pentium's time stamp counter and seeds the random number generator using this value.

1.2.1.3.16 round

```
10 i = round( f )
```

This function converts a floating point value to an integer via rounding.

1.2.1.3.17 sin

```
10 f1 = sin( f2 )
```

This function computes the sine of its parameter. The parameter must be an angle in radians. This function is most accurate when the angle is in the range -2*pi..2*pi radians.

1.2.1.3.18 sqrt

```
10 f1 = sqrt( f2 )
```

This function computes the square root of its parameter. The parameter must be greater than or equal to zero.

1.2.1.3.19 str

```
10 s1 = str( s2 )
```

This function call simply returns the value of s2 as the result.

```
20 s3 = str( s4, i)
```

The "i" parameter specifies a string length. If i is less than the length of s4, then this function simply returns s4. If i is greater than the length of s4, then this function returns a string of length i consisting of all the characters in s4 following by sufficient spaces to make the string length i characters long.

```
30 s5 = str( s6, i1, i2 )
```

This function returns a string that is i1 characters long. It consists of the first i2 characters of s6 followed by sufficient spaces to extend the resulting string to i1 characters. Note that i2 must be less than i1.

```
40 s6 = str( i1 )
```

This function call converts the integer parameter to a string using the minimum number of character positions to represent that string.

```
50 s7 = str( i1, i2 )
60 s8 = str( i1, i2, i3 )
```

These function calls convert the integer i1 to a string using at least i2 print positions. If the value of i1 would normally require fewer than i2 print positions, then STR pads the resulting string with spaces and left-justifies the string within the resulting string. If i1 requires more than i2 print positions, then STR uses however many print positions are necessary. When the first parameter is an integer expression, the STR function ignores the third parameter, if it is present.

```
70 s9 = str( f )
```

This function call converts the floating point operand to a string using exponential (scientific) notation. This form uses a field width of approximately 24 characters.

```
80 s10 = str( f, i )
```

This function call converts the floating point value f to a string using exponential (scientific) notation. The string will be i characters long. The value of i should be eight or greater to obtain reasonable results.

```
90 s11 = str( f, i1, i2 )
```

This function converts the floating point operand to a string using decimal notation. The string will be i1 characters long and STR converts the number such that it has i2 positions after the decimal point. Note that i1 must be at least two greater than i2.

In addition to the obvious use for STR (converting data to a string format), the STR function has one other really important use - it provides formatted output capabilities for the HLABasic PRINT statement. If you want to produce nice columns of numbers and other formatted output, you would typically use the STR function to achieve this, e.g.,

```
100 print "f = "; str(f, 10, 2 ), "i ="; str( i, 5 )
```

1.2.1.3.20 tan

```
10 f1 = tan( f2 )
```

This function computes the tangent of its parameter. The parameter must be an angle in radians. This function is most accurate when the angle is in the range -2*pi..2*pi radians.

1.2.1.3.21 val

```
10 n = val( s )
```

The VAL function converts a string to a numeric value. The string must contain the valid representation of an integer or real value. VAL first attempts to convert the string to an integer; failing that, it attempts to convert the string to a floating point value. If that does not succeed, the VAL raises a run-time exception. The type of the resulting numeric value depends entirely on the type of the string data.

1.2.1.4 Arrays and the DIM Function in HLABasic

In HLABasic, an array is just a special case of a data type. Unlike standard BASIC, there is no "DIM" statement for declaring arrays, instead, you call a special function (named "DIM") that returns an empty array. Therefore, to declare an array with 10 elements in it, you would use an assignment statement like the following:

```
100 x = dim( 10 )
```

This function allocates storage for 10 uninitialized variables (with indices 0..9) and associates this array with the variable x. Like most languages, you use brackets containing an integer expression to index into an array, e.g.,

```
110 x[i+2] = y*z
```

Any attempt to access an array variable without an index generates a run-time error. Likewise, any attempt to access an array element outside the range 0..n-1 (where n is the value of the DIM parameter) generates a run-time error.

Array elements are just like any other HLA variable. Array elements need not all be the same type of object; it is perfectly legal to have some integer, real, and string objects in the same array.

HLABasic doesn't directly support multidimensional array declarations. However, since arrays are just another type to HLABasic and any given array element can be any legal type, you can synthesize a multidimensional array by calling the DIM function for each element of an existing array. For example, to create a 10x10 array you could use the following BASIC code:

```
10 x = dim( 10 )
20 for i = 0 to 9
30 x[i] = dim( 10 )
40 next i
```

Since the type of each element in an array doesn't need to be the same, it is even possible to create an array that has an integer, a string, a real, and an array element, e.g.,

```
10 m = dim( 4 )
20 m[0] = 1
30 m[1] = 1.1
40 m[2] = "Hello World"
50 m[3] = dim( 4 )
```

To access an element of a synthesize multidimensional array you use syntax like the following:

```
100 m[3][2] = m[3][0]
```

1.2.2 beep

```
10 beep
```

This statement beeps the speaker for a short duration.

1.2.3 cls

```
10 cls
```

This statement clears the display.

1.2.4 color #,

```
10 color b, f
```

This statement sets the text background (b) and foreground (f) colors. The two operands can be any integer expression that evaluates to a value in the range 0..15 (since there are 16 possible foreground and background colors).

1.2.5 debug

```
10 debug
```

This statement toggles the run-time trace facility during program execution. When the trace mode is turned on, HLABasic displays the line number of each statement it executes.

1.2.6 for var = # to # {step #} / next var / next

```
10 for i = 1 to 10
    ...
90 next i

100 for i = 0 to 100 step 10
    ...
190 next i
```

Note: both the for and next keywords must appear at the beginning of the source line. There must not be any labels on these lines.

This statement implements the definite loop in HLABasic. If the STEP keyword and operand is not present, the step value is assumed to be one. If the STEP value is positive, then HLABasic executes this loop while the value of the loop control variable is less than or equal to the value of the second expression. If the STEP value is negative, then HLABasic executes the body of the loop while the loop control variable's value is greater or equal to the second expression.

The "NEXT var" statement must physically follow the FOR statement in the source file. This form of the NEXT statement is not dynamic. HLABasic assumes that all statements between the "FOR" and the "NEXT var" statements comprise the body of the loop. HLABasic will generate an error if the "NEXT var" statement does not following the corresponding "FOR" in the source file.

The NEXT statement (without a variable after the NEXT keyword) automatically restarts the last FOR statement executed. This is true even if the NEXT statement does not physically appear between the FOR and "NEXT var" statements in the program.

1.2.7 get var

```
10 get a
```

This statement reads a single character from the keyboard, creates a string object holding that single character, and assigns this string to the variable you supply as an operand. Note that this function does not wait for the

user to press enter. Execution of the HLABasic program continues the instant the user presses any key (that returns a character code) on the keyboard.

1.2.8 gosub lbl

```
10 gosub MySubroutine
```

This code pushes a "return address" onto the GOSUB stack and then transfers control to the statement associated with the given label. Control returns to the first statement following the GOSUB (possibly on the same line, after a colon following the "GOSUB LBL" statement) when the program executes a RETURN statement. See the RETURN statement for more details.

There is a built-in limit of 64 nested GOSUBs in the HLABasic language. However, this is easy to change by modifying a single constant at the beginning of the HLABasic source file.

1.2.9 goto lbl

```
10 goto targetLabel
```

This statement immediately transfers control to the specified statement in the source file.

1.2.10 if(#) then.. elseif(#) then..else..endif

The IF..THEN..ELSEIF..ELSE..ENDIF sequence provide condition execution in HLABasic. This statement allows the following syntax:

```
1000 if( expr1 ) then

   << Statements >>

2000 endif



3000 if( expr1 ) then

   << Statements >>

4000 elseif( expr2 ) then

   << Statements >>

<< Note: additional ELSEIF clauses may appear here>>

5000 endif



6000 if( expr1 ) then

   << Statements >>

7000 else

   << Statements >>
```

```
8000 endif
```

```
9000 if( expr1 ) then

   << Statements >>

9100 elseif( expr2 ) then

   << Statements >>

<< Note: additional ELSEIF clauses may appear here>>

9200 else

   << Statements >>

9300 endif
```

Important: Note that the if, elseif, else, and endif keywords must appear first on a source line. For lexical reasons, these must be the first statements on a source line.

The if and elseif clauses evaluate their expressions and execute the statements following the then keyword if the expression is true (non-zero). Otherwise they skip to the next elseif, else, or endif clause if the expression evaluates false (zero).

1.2.11 input var

This statement pauses and reads some input from the user. If the input can be converted to an integer value without error, then the input statement stores the integer result in the var variable (setting the type of var to integer). If the input cannot be converted to an integer but it can be converted to a floating point value, then the input statement converts the input to a real value and stores the result in the var object (setting its type to real). If the input is non-numeric, then the input statement converts var to a string and stores the input text in this variable.

1.2.12 gotoxy #,

This statement requires to integer expressions. It repositions the cursor in the window at the (x,y) coordinate specified by the two operands. E.g.,

```
gotoxy 5,10
```

repositions the cursor on line 10, column 5.

1.2.13 on # goto lbl_list

The on..goto statement transfers control to one of several different table based upon the value of some expression. The syntax for this statement is

```
10      on expr goto lbl0, lbl1, lbl2, lbln-1
```

This statement evaluates the expression, which must produce an integer value. If the result is zero then this statement jumps to the first label in the list following the goto keyword. If the result is one, then this statement

jumps to the second label, etc. If the value is less than zero or greater than n-1, this this statement falls through to the next physical statement in the program (i.e., the one following the on..goto statement).

1.2.14 on # gosub lbl_list

This statement is very similar to the on..goto statement except it calls the subroutine at the specified label rather than jumping directly there. When the subroutine returns, it returns control to the first statement beyond the on..gosub. Example:

```
10      on i-2 gosub sub1, subA, subB
```

1.2.15 print expr_list

The print statement displays the data specified by the expression list on the console output device. The expr_list item above represents a list of zero or more HLABasic expressions, each separated by a comma or a semicolon. Items that are separated by semicolons are printed adjacent to one another, items separated by commas are separated by a tab field. HLABasic automatically formats the expressions according to their data type and prints the value of the expression in a string format. Example:

```
10      print "Hello World, i=", i
```

Note that if the statement ends with a semicolon, HLABasic does not print a newline sequence after the textual output. If the statement does not end with a semicolon, the HLABasic will print a newline sequence after displaying the associated text.

HLABasic's print statement does not directly support formatted output, but you can use the STR function to format the data on output. See the description of the STR function for more details.

1.2.16 readln var

This reads a line of text from the user and stores that text into the variable specified as this statement's operand. The operand is always converted to a string variable, even if the input takes the form of an integer or a floating point value (this is the difference between readln and input). Example,

```
10      readln inputVar
```

1.2.17 rem

This statement allows you to insert a comment (remark) into the code. The remainder of line is ignored by the interpreter. Note that a colon (":") character does not end the comment, only the physical end of the line does this. Example,

```
10      rem this is a comment
```

1.2.18 return

Returns from a subroutine called by GOSUB. See GOSUB and ON..GOSUB for more details. Example:

```
10      return
```

1.2.19 stop

Stops the program and displays the current line number. Example,

```
10      stop
```

1.2.20 wait

Delays the program for the specified number of milliseconds. Example:

```
10 wait 2000 : rem delays for two seconds.
```

1.2.21 Statements and Other Things That Would be Nice to Have

read..data
file I/O
while..endwhile
character sets
parameters
user-defined functions