Implementation of 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.

This document provides a brief roadmap to the internal operation of the HLABasic interpreter. If you are interested in modifying HLABasic for any reason, you will probably want to read this document before attempting such modifications.

HLABasic is written in assembly language using the High Level Assembler (HLA) which is, in fact, the source of HLABasic's name. This document assumes that you are already familiar with HLA syntax and are comfortable using the HLA compiler. If you need information about HLA, please consult the appropriate documentation at http://webster.cs.ucr.edu.

The HLABasic interpreter can be grossly divided into five modules: the command interpreter, the tokenizer/parser, the detokenizer, the compiler, and the run-time interpreter. This document is organized along these lines, discussing each of these major parts of the interpreter in separate sections.

The HLABasic interpreter was written to be an example of a fair sized program written in HLA. The code was written to be easy to understand and modify, not as an example of the fastest interpreter one could possibly write. Nor was any attempt made to make the code as short as possible. Those looking for a very efficient interpreter should probably look elsewhere (or be prepared to spend some time optimizing this code). This is not to suggest that HLABasic is particularly slow or large. HLABasic uses good algorithms in the interpreter, so it is not slow by any means; it does quite well against other interpreters written for the 80x86. The interpreter's EXE file is about 76K on the disk (including lots of data, this isn't all just code). At run-time, the interpreter consumes a bit more memory to hold the BASIC program and any variables it uses. So although it's not tiny, it's not a complete bloat either. With a bit of work, one could easily double the speed of the interpreter and probably cut the size of the interpreter in half. However, that task is left to someone who is interested in the challenge of doing so. Most people who have 256 MBytes of RAM on their machine don't really care if the interpreter is 50K or 80K. As for possible speed improvements, someone who really wants a faster interpreter should consider using a compiler like Visual BASIC is performance is paramount.

HLABasic is public domain. So feel free to do whatever you want to do with this code. One possible use for this interpreter is to embed it in other products to help create a BASIC-like scripting language for that product. Whatever, feel free to do as you wish with this code. As noted above, this code has not been tested, so you must take full responsibility for all use, abuse, and mis-use of this code.

HLABasic is a traditional console-based version of the language. Don't expect fancy graphics facilities or a GUI rapid application development tool like Visual BASIC. Although it's relatively easy to add new features to HLABasic, you shouldn't attempt to turn it into VB (Visual BASIC). That would be far too much work given the availability of languages like VB, Delphi, and Java.

1.1 HLABasic Organization and Overview

As noted earlier, HLABasic can be divided into about five main sections (plus some change). This section will quickly list all the procedures in HLABasic and point out the section to whom those procedures belong. This will provide an overview of the whole system.

Table 1

Procedure	Section	Description
FreeVars	Interpreter	Garbage collection routine. Used to free up storage used by a variable whenever the program assigns a new value to that variable.

Table 1

Procedure	Section	Description
GetLineNum	Command	Verifies that an input line number is correct.
AtLineNumber	Interpreter	Used to print the line number whenever a run-time error occurs.
AllocCons	Command	Used to initialize the standard I/O file handles whenever the program executes.
RedimWindow	Command	Used to create a new console window for output.
WinEdit	Command	Initializes the console display (colors, etc.)
CLS	Interpreter	Clears the console window.
InitP	Interpreter	Called by the interpreter to initialize the GOSUB and FOR stacks.
FindLine	All	Locates a specific line in the source file by line number.
Compile	Compiler	Makes a couple of passes over the source code when the user issues the RUN command to patch up addresses and other optimization-oriented features.
DoDebug	Interpreter	Prints the line number for each statement at run time if the trace mode is turned on.
RealToStr	Interpreter	Converts a real number to a string using exponential or decimal form, depending on the size of the value (used by STR and PRINT, for example)
StrToNumber	Interpreter	Converts a string to an integer or real number.
NumberToStr	Interpreter	Converts a string to an integer or a floating point value, depending upon the number's format.
MakeInt	Interpreter	Tries to coerce its parameter to an integer.
Expr	Interpreter (this is the expression evaluator)	Evaluates an arithmetic expression.
MakeCompatible	Interpreter / ExprEval	Attempts to coerce two operands so that they are the same type.
Factor	Interpreter / ExprEval	Handles IDs, constants, parenthetical expressions, functions, and unary operators in an arithmetic expression.
MULx	Interpreter / ExprEval	Handles the *, /, and % operators in an arithmetic expression.
ADDx	Interpreter / ExprEval	Handles the + and - operators in an arithmetic expression.
RELx	Interpreter / ExprEval	Handles the relational operators (<, <=, =, <>, >=, and >) in arithmetic expressions.
ANDx	Interpreter / ExprEval	Handles logical expressions involving the AND operator.

Table 1

Procedure	Section	Description
ORx	Interpreter / ExprEval	Handles logical expressions involving the OR operator.
Deallocate	Interpreter	Deallocates storage held by a variable at run-time.
ProcessIndex	Interpreter	Processes array indexes of the form "[expr]" after an array variable.
RunProc	Interpreter	Handles the RUN command - starts the execution of the program. This procedure contains the code for the actual interpreter portion of HLABasic.
Detokenize	Detokenizer	Converts the tokenized source format to text in response to the LIST command.
ListProc	Command	Handles the HLABasic LIST command.
ParseLine	Tokenizer	Converts a textual source line to its tokenized form. Also reports any syntax errors on the line.
PutToken	Tokenizer	Emits an HLABasic token to the tokenized source array in memory.
Expression	Tokenizer	Tokenizes an arithmetic expression.
SkipSpcs	Tokenizer	Skips over any whitespace on the input line during tokenization.
matchNeg	Tokenizer	Handles the unary minus sign encountered while tokenizing an expression.
matchNot	Tokenizer	Handles the NOT operator while tokenizing an expression.
GetParenExpr	Tokenizer	Matches and tokenizes an expression surrounded by parentheses.
GetBracketExpr	Tokenizer	Matches and tokenizes an expressions surrounded by brackets.
matchID	Tokenizer	Matches an identifier or function name encountered in an expression.
MatchIntConst	Tokenizer	Matches and tokenizes a literal integer constant encountered in an expression.
matchFltConst	Tokenizer	Matches and tokenizes a literal real constant encountered in an expression.
matchTerm	Tokenizer	Matches IDs, constants, function calls, and parenthetical expressions in an expression.
ParseID	Tokenizer	Matches and tokenizes identifiers appearing in source code.
ParseStmt	Tokenizer	This is the main tokenizer routine. It accepts a line of text, parses it (checks for correctness), and converts that text to a string of tokens in the source file array.
DumpProc	Command	This procedure handles the DUMP command.
EditProc	Command	Handles the EDIT command.
MiniBasic	Command	Main program and command processor.

1.2 The Command Processor (MiniBasic - Main Program)

When HLABasic first begins execution, it initializes a bunch of variables that the interpreter uses. See the source code for details on this. One important initialization, however, is storing a copy of the ESP register in the Sstack variable. The reason for doing this is so that the system can reset the stack pointer if some sort of exception or other situation causes the interpreter to restart after some sequence of operations that can leave the stack in an indeterminate state. Then the program enters a (really big) infinite loop that processes commands. At the bottom of this loop, the interpreter restores ESP from the Sstack variable in case some premature command abort left some junk on the stack.

Just inside this big infinite loop is a TRY..ENDTRY statement. Many abnormal operations (like a run-time error or a command syntax error) raise an exception that returns control back to the main processing loop. This, effectively, halts whatever operation was in progress (including an executing program) and returns control back to the main command interpreter (e.g., stopping a running program that had a run-time error). See the EXCEPTION clauses in this TRY..ENDTRY statement for details on the types of exceptions the interpreter generates.

Once inside the TRY..ENDTRY statement, the interpreter prompts the user for a command and then reads a line of text from the user. The command processor uses the HLA Standard Library pattern matching facilities (i.e., that pat.match function) to determine which command the user has entered. If the command begins with a decimal integer, then the command interpreter checks to see if it should delete a line of text, edit an existing line of text, or insert a new line of text into the source file. Other patterns check for the DUMP command, the LIST command, the EDIT command, the RUN command, the NEW command, the DEBUG command, the LOAD command, the SAVE command, or the QUIT command. If the user enters anything else, then the command processor complains about an illegal command.

If you want to add a new command to the language, then you will probably want to add a pat.alternate clause to the main pat.match..pat.endmatch statement in the main program. If you are unfamiliar with how the HLA Standard Library pattern matching module works, you'll probably want to read up on this very interesting module in the HLA Standard Library documentation. You can also use the code for a similar command as a template for any new commands you add to the interpreter.

The LOAD and SAVE commands are probably the most interesting of the bunch. The SAVE command opens a text file (with the user-specified file name), redirects the standard output to this file, and then executes the LIST command in order to dump a text listing of the program to the specified text file. The LOAD command does the converse - it redirects the standard input from a user-specified file and then tokenizes each input line read from the file, exactly as though the user were typing these lines in at the keyboard.

Although the HLA pattern matching library is not blazing fast, do not be concerned about its performance for the command processor. This is not a time-critical component of the program, so the fact that faster ways to parse a command may exist is really irrelevant here; most commands execute instantaneously as far as the user is concerned, so investing effort in making this code run faster is a waste of time.

1.3 The Tokenizer

When the user enters a statement that begins with a line number, it is the job of the Tokenizer code to take this input line, in text form, verify its correctness, convert it to the equivalent token sequence, and insert these tokens into the source file at the appropriate place. The ParseStmt procedure handles this operation. So if you want to add a new statement to the HLABasic language, the ParseStmt procedure is the first procedure you need to modify (there are others, we'll get to those in a little bit).

To add a new statement to the tokenizer, you must add a pat.alternate section to the pat.match..pat.endmatch statement in the ParseStmt function. This pattern matching code must check the syntax of the statement. If the syntax is correct, the code should emit a sequence of byte tokens to the LineBuffer array at the offset specified by the index variable (updating index to point beyond any tokens you add to the LineBuffer array). If you're creating a brand new statement with its own keyword(s), then you'll need to add the symbol for the keyword's token to the ReservedWords enumerated data type at the beginning of the source file. The easiest way to add a new statement to HLABasic's tokenizer code is to find some other statement whose syntax is similar to the statement you wish to create and use the code for that other statement as a template for your new statement.

The ParseStmt tokenizer code calls a couple of helper functions to parse items appearing within a statement. The best examples are the Expression procedure and the ParseID procedures. These two procedures parse and tokenize arithmetic expressions and HLABasic identifiers. So, for example, if you want to add new operators to HLABasic, you'd modify the Expression procedure.

The ParseStmt code is not time critical. This procedure executes immediately after the user presses the enter key after entering a new statement. On most modern machines, the parsing and tokenization occur so rapidly that the system comes back nearly instantaneously. Hence, there is no need to try to write optimized code.

1.4 The Detokenizer

The Tokenize procedure is responsible for taking HLABasic statements as input from the user, verifying the syntax of these statements, and translating them to a tokenized form in memory. The Detokenize procedure has the converse responsibility - it's job is to convert the tokenized statement in memory back to a string. HLABasic uses this facility for two operations: listing the source file and saving the source file to disk. Therefore, the second step you must do when adding a new statement to the HLABasic language is to modify the Detokenize procedure so that it can detokenize your new statements after Tokenize has converted them to tokens.

The Detokenize procedure is probably the easiest to modify. It's really little more than a giant loop that moves through a statement, picks off a byte, and uses that byte to select a CASE in a SWITCH statement. Each CASE in this SWITCH statement appends a corresponding string to the deststr variable. Note that the cases don't have to worry about the structure of a whole statement – they've only got to deal with a single token at one time. This is what makes the Detokenize procedure so easy to modify. Typically, when you add a new statement to HLABasic, you've only got to add one CASE to the SWITCH statement that translates your new token to the corresponding keyword for your new statement.

The Detokenizer procedure is not particularly time critical. Only the LIST and SAVE commands use this code. For the LIST operation, speed is irrelevant since the computer converts the tokenized code to text data many orders of magnitude faster than you can read it. For the SAVE command, the speed of the Detokenize procedure can have a negative impact whenever the user is saving really large BASIC files to disk. However, the Detokenize code isn't particularly slow, so the SAVE operation is virtually instantaneous for most files.

1.5 The RunProc Procedure

The RunProc procedure handles the run-time interpreter portion of the program. This is the one section of the interpreter that must be fast since any inefficiencies in this code will result in a slower executing BASIC program. This is not to suggest that RunProc is highly optimized (it is not), it's just a gentle warning to be careful about the code you write or modify in this procedure (and related procedures) since it can affect the run-time performance of your BASIC code.

The run-time code is broken down into two components: a "compiler" phase and an interpreter phase. Whenever you issue the RUN command, HLABasic first makes a couple of quick passes over the source file and related tables. First, the code scans through the source file to find statement labels. For each statement label it finds, it copies the address of that source line into the symbol table entry for that statement label. By doing this, the interpreter doesn't have to scan through the file looking for labels at execution time. This is a huge performance optimization, especially for larger programs.

One last step that the compiler does is to make another pass through the source file and discover where the "context free" statements begin and end. HLABasic currently supports two context-free statements: IF..ELSEIF..ELSE..ENDIF and FOR..NEXT. During this phase, the compiler procedure locates the corresponding ending statements for the IF and FOR statements (e.g., ELSEIF, ELSE, and ENDIF for IF and NEXT for FOR). The compiler stores the address of the terminating statement in a special token field of the IF or FOR statement. That way, if the IF expression evaluates false or the FOR loop control variable is out of range, control can transfer directly to the ending clause for that statement. Again, this is an optimization that saves considerable time during interpretation.

After the compile phase patches up the destination addresses, the interpreter begins running. Note that the compile phase is very quick. Few users will ever notice a delay between the point they type RUN and the program begins execution, even for larger BASIC programs.

The interpreter itself (RunProc) is basically a giant SWITCH statement that fetches tokens from the source file and transfers control to a CASE that executes the statement associated with that token. If you're adding a new statement to the language, the last step you must do (after fixing ParseStmt and Detokenize) is to add an appropriate CASE to RunProc to actually execute the statement.

Within the SWITCH statement in RunProc, the EBX register always points at the beginning of the current line that is executing. The ECX register points at the current token within the line that the interpreter is executing. Each CASE adjusts ECX, as appropriate, after executing the code for the current token, so that ECX points at the next token to process. Again, the easiest way to add a new statement to the language is by finding a statement with similar syntax and semantics and copying the code for that statement.

The expression evaluator in the interpreter is, perhaps, the weakest component of the HLABasic interpreter. The expression evaluator uses a traditional "top-down recursive descent" parsing algorithm to evaluate expressions. This scheme is not very high performance. A better solution would be to modify the ParseStmt code to store the tokens in a reverse-polish format. This would speed up the evaluation of expressions at run-time by leaps and bounds (it would make detokenizing the code a bit more difficult, but it would still be worth it). This is definitely a change that needs to be made to the interpreter at one point or another. Although there are no profile tests to prove this, it is probably the case that expression evaluation is the slowest part of the interpreter and probably has a very negative impact on the overall performance of the interpreter.

Probably one of the most common extensions that will occur to HLABasic is the addition of new functions to the language. Adding a new function is really quite easy. Just add a keyword to the ReservedWords table, add the syntax for the function call to the ParseStmt/matchID/Expression code, make a simple addition to Detokenize, and then add the code for your new function to the Factor function in the run-time expression evaluator code.

1.6 Wrapping it Up

HLABasic, while larger than many assembly language programs at slightly more than 10,000 lines, isn't a tremendously huge program (compare this, for example, with the HLA compiler that is in excess of 100,000 lines of code). Furthermore, the code is structured reasonably well and contains a decent amount of comments. Therefore, the next thing for the adventuresome explorer of HLABasic is to just sit down and start reading the code. After playing around with the code, you'll begin to see how it works.

One word of caution - the only truly non-intuitive stuff in the interpreter is how errors are handled. Remember, there's a big TRY..ENDTRY block around the main program. Anytime the interpreter (or some other part of the system) has a run-time exception, the program raises an exception that transfers control back to the main program (from wherever and however deep the exception occurred). Once you get past this, and manage to master the HLA Standard Library's pattern matching modules, the rest of the code is fairly easy to figure out by someone who's had a little compiler theory.

Good Luck.