ScriptBasic Developers Manual

Peter Verhas

Short Contents

Table of Contents

1 Introduction

ScriptBasic is an open source scripting implementation of the programming language BASIC. The language and the implementation make the software a considerable choice in many cases when there is a need for some scripting tool. This software can be used where traditionally Perl, Python, TCL or some other scripting interpreter is used.

The language itself is BASIC with powerful command set and large number of extension modules. The language was designed so that this is familiar for all BASIC programmers no matter which dialect one has experience with. Thus the learning curve of the language is very steep: you can start to write your first programs just in few minutes.

The interpreter is built modular, well documented, and easy to read source code and has well defined and documented interfaces. It is easy to embed the interpreter into an application and it is also easy to write external modules extending the language. Applications embedding ScriptBasic can build virtual machines in multiple or in single process running interpreters simultaneously independent of each other, but they can also execute multi-thread applications running interpreter threads that provide features for the BASIC programs running parallel to communicate to each other.

This means that ScriptBasic can be the language of choice in situations when the application programmer wants to implement some programmability feature into the application. It will attract the users of the application because there is no need to learn a new programming language at the time when they have to learn the functions of the application anyway. The programmer on the other hand has an easy job to integrate ScriptBasic into the application because it was designed for the purpose.

To sum up these in a listing the language and the implementation has the following features:

IT IS BASIC. No question, this is the MOST important feature of ScriptBasic. There are a lot of people who can program BASIC and only BASIC. There are many people, who can not really program. Those who do not really know what programming is, and still: they write their five-liners in BASIC to solve their simple problems. They never write Perl, Tcl, Java or C. Therefore it is BASIC.

SCRIPTING language. There are no data types in the language. You can store real numbers, integer numbers and strings in any variable. You can mix them and conversion is done automatically.

PORTABLE Available in C source and can be compiled on UNIXes as well as on Windows NT.

4E LANGUAGE, which means easy to extend, easy to embed. ScriptBasic was developed to provide clean and clear interfaces around it, and inside it. It is easy to embed the language to an application and use it as a macro language just like TCL. It is also easy to implement new built-in function and new commands. You can develop dynamically loaded libraries that ScriptBasic may load at run time. The language source is clean, well documented and development guides are on the way.

COMPILED CODE ScriptBasic creates intermediate compiled code, which is interpreted afterwards. Syntax analysis is done at first and only syntactically perfect programs start to run. The compiled code is put into a continuous memory space and compiled code can be saved and loaded again to run without recompilation. This is vital for CGI scripts and is not available for most scripting programming languages. Compiled code is binary, not readable. Therefore you can develop and distribute programs and getting some help to protect your intellectual property. You need not give the source code.

MULTI THREAD aware. Although the current implementation is not multi thread, all the code was designed to be thread safe. You can embed the code into systems that run multiple interpreters in the same process. On the other hand the interpreter can run the same code in multiple threads and was designed to be capable handling call-back functions, and multithread programs in the future.

DEBUGGER The BASIC programs can be debugged using the external debugger.

1.1 Chapters

This documentation has five main chapters. These are

See ⟨undefined⟩ [Interpreter Architecture], page ⟨undefined⟩ describes the overall architecture of the interpreter, lists the different modules, and contains the reference documentation of the C functions implemented in the individual modules.

See ⟨undefined⟩ [Embedding the Interpreter], page ⟨undefined⟩ describes how to write an application that embeds The ScriptBasic interpreter. This chapter also contains the reference documentation of the full C API.

See ⟨undefined⟩ [Extension Modules], page ⟨undefined⟩ describes what extension modules are, how to use them and how to write extension modules that may provide access to special programs, operating system features, and are not implemented in the interpreter core.

See Chapter 5 [Preprocessors], page 51 detail how to write preprocessor extensions to ScriptBasic. These extensions provide development functionalities like debugging, profiling and processing of the source code in addition to the processing of the interpreter.

See ⟨undefined⟩ [Compilation], page ⟨undefined⟩ details how to compile ScriptBasic under Windows NT.

I advise you to read the architecture chapter first to get the overall picture. After that you can read one of the three middle chapters based on your plans. If you plan to embed the interpreter, read the second chapter. If you want to write an extension module, read the third chapter. If you intend to write a preprocessor, read the fourth chapter. You may want to do more than one even.

By now you may not need what the differences are. This will change after reading the architecture chapter.

The last chapter is to be read just before you start to experiment with ScriptBasic. This will give you guide lines how to compile the program from source. This may not be needed for extension module and preprocessor developers, but it does not do any harm either. It may help.

2 Interpreter Architecture

This chapter tells you the architecture of the interpreter. It is not a must to read this chapter, and you may find that some topic is irrelevant or not needed to learn to embed or to extend ScriptBasic. However understanding the internal working order of ScriptBasic should help you understand why some of the extending or embedding interfaces work the way they actually do. So I recommend that you read on and do not skip this chapter.

To read this chapter and to understand the internal working of the interpreter is vital when you decide to write an internal preprocessor. Internal preprocessors interact with not only the execution system but also the reader, lexer, syntaxer and builder modules along the way one after the other as they do their unique job processing a BASIC program, thus internal preprocessor writers have to understand how these modules work.

ScriptBasic is not a real scripting language. This is a mix of a scripting language and compiled languages. The language is scripting in the sense that it is very easy to write small programs, there is no need for long variable and function declarations. On the other hand the language is compiled into an internal code that is executed afterwards. This is the same or similar technique, which is used in the implementations of the language Java, Perl, Python and many other languages.

ScriptBasic as a language is a BASIC dialect that implements most BASIC features that BASIC implementations usually do. However ScriptBasic variables are not typed, and dynamic storage, like arrays are automatically allocated and released. A ScriptBasic variable can store a string, an integer, a real value or an array. Naturally a variable can not store more than one of any of these types of values. But you need not declare a variable to be INTEGER or REAL, or STRING. A variable may store a string at a time and the next assignment command may release the original value and store a different value in the variable.

When a program is executed it goes through several steps. The individual steps are implemented in different modules each being coded in a separate C language source file. These modules were developed so that they provide clear interface and thus could be replaced. The lexical analyzer uses the functions provided by the reader, the syntax analyzer uses the functions provided by the lexical analyzer and so on. The modules never dig into each others private area.

The modules are listed here with some explanation.

EXTERNAL PREPROCESSOR

This module executes external preprocessors. These preprocessors are standalone executable programs that read the source program and create another file that is read and processed by the ScriptBasic interpreter. If an external preprocessor is used the source file is usually not BASIC but rather some other language, usually a BASIC like language, which is extended some way and the preprocessor creates the pure ScriptBasic conformant BASIC program. The sample preprocessor supplied with ScriptBasic is the HEB (HTML Embedded BASIC) preprocessor that reads HTML embedded BASIC code and creates BASIC program. This HEB source file is a kind of HTML with embedded program fragments, which you may be familiar with in case you program PHP or Microsoft BASIC ASP pages. The HEB preprocessor itself is written in BASIC and is executed by ScriptBasic. Thus when a HEB "language" is executed by ScriptBasic it starts a separate instance of the interpreter and executes the HEB preprocessor on the source file. Of course the HEB preprocessor could be implemented in any language that can be compiled or some way executed on the target machine. Actually the very first version of the HEB preprocessor was written in Perl so when it was first tested the ScriptBasic interpreter started a Perl interpreter before reading the generated BASIC code.

Note that the HEB preprocessor provided in the ScriptBasic package is an example implementation and lacks many features. It can, for example, be fooled by putting a `%>` characters into a BASIC string constant.

READER

This module reads the source file into the computer memory. Usually source programs are not too big compared to computer memory and thus can be read into the operational memory (RAM). ScriptBasic source code is approximately 1MB and I develop it on a station that has 386MB memory. This means that even a fairly large program can fit into the memory seamlessly. BASIC programs executed by the ScriptBasic interpreter are likely to be much smaller than that.

The source code is stored in memory pieces that form a linked list. Each element of the list contains one line of the source code and the information of the line for debugging and error reporting purposes. This information includes the file name that the line was read and the line number. Later when the lexer (detailed later) performs lexical analysis it will inherit this information and when there is a lexical or syntactical error the line number is reported correct.

The reader module also handles the `include` and `import` directives that are used to include files into the source file. (Note that `import` inserts the content of the file only if it was not loaded yet.)

The module also processes the lines that look

```
use preprocessor
```

and loads the internal preprocessor named on the line. See Chapter 5 [Preprocessors], page 51

When the module is ready the latter modules have the full source file in memory ready to be processed. The module also provides `getc` and `ungetc` like functions to get the read characters one by one. These are is used by the lexer.

LEXER

The lexer module uses the line stream (or the character stream if we view it from a different point of view) provided by the reader. It reads the characters and builds up a linked list. Each element of the list contains a token, like BASIC keyword, a real or integer number, symbol, string, multi-line string, or character. The list of tokens is stored in a form of linked list in the order the tokens appear in the input. Each element also contains extra information about the token that identifies the name of the file and the line number inside the file where the token originally was.

When the lexer is finished the list of lines is not really needed any more and the reader is ready to release the memory occupied by the source lines read into memory.

The lexer also provides functions that are used by the syntax analyzer to read the tokens in sequence one after the other as needed by the syntax analysis.

SYNTAXER

The syntaxer reads the list of tokens provided by the lexical analysis module and creates an internal structure that is already very similar to the executable internal code of ScriptBasic. The syntax analyzer finds any programming error that is not syntactically correct and when it is ready the result is a huge, cross-linked memory structure that contains the almost-executable code.

The syntax analyzer is responsible building up the evaluation trees of the expressions, the execution nodes, variable numbering and so on.

When the code refers to a variable named for example `variable` the syntax analyzer is responsible to allocate a slot for the variable and to convert the name to a serial number that identifies the variable whenever it is used. Beyond the syntax analyzer there are no named variables anymore (except in case of debuggers). There are global variables listed from `1` to `n` and local variables also listed by numbers. There are also no names for the functions. Each function is identified by a C pointer to the node where the function starts.

To ease the life of those who want to embed ScriptBasic the symbol table that list the global variables and the functions and subroutines is appended to the byte-code and there are functions in the `scriba_*` embedding interface that handles these symbol tables. However ScriptBasic itself does not use variable or functions/subroutine names beyond the syntax analyzer.

BUILDER

The builder is the module that creates the code, which is used by the execution system. Why do we have a separate builder? Isn't it the role of the syntax analyzer to build the code?

Yes, and no. The code that was created by the syntax analyzer could be used to execute the BASIC program, but ScriptBasic still inserts an extra transformation before executing the program. The reason for this extra step is to create a byte code that can be stored in a continuous memory area and thus can easily be saved to or loaded from disk.

When the syntax analyzer creates the nodes it does not know the actual number of nodes of the byte-code, nor the number of different strings, or size of the string table. While the code is created the syntax analyzer allocates memory for each new block it creates one by one. The nodes are linked together using C pointers. This means that the final memory structure is neither continuous in memory nor can be saved or loaded back to disk.

When the builder starts the number of the nodes just as well as the total string constant size is known. The builder allocates the memory needed for the whole code and fills in the actual code. The node size is a bit smaller than that of the syntax analyzer and they refer to each other using node serial numbers instead of pointers. This is almost as efficient as using pointers and the actual value does not depend on the location of the node in memory and this way the code can be saved to disk and loaded again for execution.

EXECUTOR

The executor kills the code. Oh no! I am just kidding.

It actually executes the code. It gets the code that was generated by the module builder and executes the nodes one by one and finally exits.

The following sections detail these modules and also some other modules that help these modules to perform their actual tasks.

2.1 External Preprocessor

The module that implements the external preprocessor handling is coded in the C source file '`epreproc.c`' (Note that there is a file '`ipreproc.c`' that handles internal preprocessors.) The sole function in this file is named `epreproc`. This is quite a complex function that gets a lot of arguments listing all the preprocessors that have to be executed one after the other on the source file or on the file created by the previous preprocessor in case there are more than one preprocessors to be applied in chains. The function gets a series of preprocessors to be executed in the command line but in case there is no preprocessor defined in the arguments it executes the preprocessors that have to be executed according to the BASIC program file name extension and the configuration file.

Only the preprocessors that are configured can be executed. Each preprocessor has a symbolic name, which is used to name it in the configuration file or on the command line using the option '`-p`'.

An external preprocessor is a program that reads a text file and produces a text file. ScriptBasic executes the external preprocessor in a separate process and supplies it with the input and the output file name on the command line. For example the '`scriba.conf.lsp`' configuration file may contain the following lines

```
preproc (
  extensions (
     heb "heb"
     )
  external (
    heb (
      executable "/usr/bin/scriba /usr/share/scriba/source/heber.bas"
      directory "/var/cache/scriba/hebtemp/"
      )
    )
  )
```

The key `executable` defines the executable image of the preprocessor. In this case this is an executable image and an argument because the HEB (HTML Embedded Basic) preprocessor is written in BASIC. The other arguments, namely the input file name and the output file name are appended after this string separated by space. When the preprocessor is executed the actual command line is

```
/usr/bin/scriba /usr/share/scriba/source/heber.bas myprog.heb /var/cache/scriba/heb
```

(The above sample command line may be split into two lines by the printing/displaying system, but this is essentially a single command line with the executable name and the two arguments.)

The final argument is a file name automatically generated by ScriptBasic using MD5 calculation of the input file name full path (may and presumably should not be correct in this example above as I just typed some random 32 characters). The preprocessor should read the file `myprog.heb` and should generate `/var/ca...MN` (forgive me for not typing that long file name again).

If there are more preprocessor to be executed on the generated result then the next is executed on the generated file as input file and another output file is generated.

The preprocessor program should gracefully exit the process it runs in and exit with the code `0`. Any other process exit code is treated as error and the further processing of the BASIC program is aborted.

The sample code for the HEB preprocessor can be found in the ScriptBasic source distribution. Note that this preprocessor implementation is a sample and is not meant to be a professional, commercial grade preprocessor.

2.2 Reader

The module reader is implemented in the C source file '`reader.c`' This is a very simple module it just reads the lines from the source code files and stores the actual text in memory using linked lists. There is not too much possibility to configure this module except that the memory handling functions and the file opening, closing and reading functions are used via function pointers that can be altered by the caller.

These input modules configurable by the embedding application make it possible to read the BASIC source code from database, network or from some other stream not being conventional text file.

Like any other module in ScriptBasic the reader module uses a module object. This is like a class definition except that the interpreter is coded in C and thus there is nothing like VTABLE or inheritance. Otherwise the code is object oriented. Here we list the actual definition of the reader object. Note however that this is actually a copy of the actual definition from the file '`reader.c`' and it may have been changed since I wrote this manual. So the reader object by the time I wrote this manual (v2.0.0) was:

```
#define BUFFER_INITIAL_SIZE 1024 //bytes
#define BUFFER_INCREMENT 1024 // number of bytes to increase the buffer size, when
  char *Buffer;  // buffer to read a line
  long dwBuffer; // size of Buffer in bytes
  long cBuffer;  // the number of character actually in the buffer

  pSourceLine Result; // the lines of the file(s) read

// iteration variables
  pSourceLine CurrentLine; // the current line in the iteration
  long NextCharacterPosition; // the position of the next character to be returned
  char fForceFinalNL; // if this is TRUE then an extra new line is
                      // added to the last line if it was terminated by EOF█

  pReportFunction report;
  void *reportptr; // this pointer is passed to the report function. The caller sho
  int iErrorCounter;
```

```
    unsigned long fErrorFlags;

    pImportedFileList pImportList;

    char *FirstUNIXline;
    struct _PreprocObject *pPREP;
     ReadObject, *pReadObject;
```

The pointers `fpOpenFile`, `fpGetCharacter` and `fpCloseFile` point to functions that are used to open the input file. The pointer `pFileHandleClass` set by the higher code using the module reader is passed to these functions without caring its meaning. This is not used by the standard file input/output functions that are used by the command line version of the program, but can be useful for program environment when the source file is stored in some other forms and not in a file. An example of such use can be seen in the function `scriba_LoadProgramString` implemented in the file '`scriba.c`'.

The linked list of source lines is stored in the structure named `SourceLine` The definition of this structure is

```
  typedef struct _SourceLine
    char *line;
    long lLineNumber;
    long LineLength;
    char *szFileName;
    struct _SourceLine *next;
     SourceLine, *pSourceLine;
```

You can see that each source line is pointed by the field `line` and the length of the line is also stored. The reason for this extra field is that the line itself may contain zero character although this is rare for a program source file to contain zero character inside.

Before the file is read the function `reader_InitStructure` should be called. This is usual for the ScriptBasic modules. This function initializes the reader object to the usual values that actually ScriptBasic needs.

The reader provides function `reader_ReadLines` that actually reads the lines and also processes all lines that contain an `include` or `import` directive to include a line.

The reader has some extra functions that are specific to ScriptBasic or generally saying are specific to program source reading.

Source programs under UNIX usually start with a line

```
#! /usr/bin/scriba
```

that tells the operating system how to start the code.

((((Some very old and totally outdated version of some UNIX systems check the first four characters to look for `#! /`. There is a space between the `!` and the `/`. So if you want to be look a real code geek put this extra space before the executable path. To be honest I have never encountered this issue since 1987 when I met my first UNIX at TU Delft, Hollandia. OK that's for the story, get back to the reader!))) Always close the parentheses you open!)

The reader recognizes this line if this is the very first line of the program and unlinks it from the list. Instead you can reach this line via the reader object variable `FirstUNIXline`.

This is specific to program source reading and not general file reading. But the reader module is a program source reader or more specific: a BASIC, even ScriptBasic program source reader, though it was coded to be as general as possible.

Another specific issue is the new line at the end of the last line. Lines are usually terminated by new-line. This line terminating character is included in the string at the end of each element of the linked list the reader creates. However when the last line is terminated by the pure EOF some syntax analyzers may fail (ScriptBasic syntax analyzer is also an example, but the reader is kind to care about this). For the reason if the variable `fForceFinalNL` is set to be TRUE this line gets the extra new-line when read.

2.3 Lexer

The module lexer is implemented in the C source file '`lexer.c`' This is a module that converts the read characters to a list of tokens. The lexer recognizes the basic lexical elements, like numbers, strings or keywords. It starts to read the characters provided by the reader and group it p into lexical elements. For example whenever the lexical analyzer sees a `"` character it starts to process a string until it finds the closing `"`. When it does the module creates a new token, links it to the end of the list and goes on.

To do this the lexical analyzer has to know what is a keyword, string or number.

Because general purpose, table driven lexical analyzers are usually rather slow Script-Basic uses a proprietary lexical analyzer that is partially table driven, but not so general purpose as one created using the program LEX.

There are some rules that are coded into the C code of the lexical analyzer, while other are defined in tables. Even the rules coded into the C program are usually parameterized in the module object.

Lets see the module object definition from the file '`lexer.c`' (Note that the C `.h` header files are extracted from the `.c` files thus there is no need to double maintain function prototypes.)

Note however that this is actually a copy of the actual definition from the file '`lexer.c`' and it may have been changed since I wrote this manual. So the lexer object by the time I wrote this manual was:

```
typedef struct _LexObject
  int (*pfGetCharacter)(void *);
  char * (*pfFileName)(void *);
  long (*pfLineNumber)(void *);
  void *pvInput;
  void *(*memory_allocating_function)(size_t, void *);
  void (*memory_releasing_function)(void *, void *);
  void *pMemorySegment;

  char *SSC;
  char *SCC;

  char *SFC;
  char *SStC;
```

```
        char *SKIP;

        char *ESCS;
        long fFlag;

        pReportFunction report;
        void *reportptr;
        int iErrorCounter;
        unsigned long fErrorFlags;

        char *buffer;
        long cbBuffer;

        pLexNASymbol pNASymbols;
        int cbNASymbolLength;

        pLexNASymbol pASymbols;

        pLexNASymbol pCSymbols;
        pLexeme pLexResult;
        pLexeme pLexCurrentLexeme;
        struct _PreprocObject *pPREP;
        LexObject, *pLexObject;
```

This `struct` contains the global variables of the lexer module. In the first "section" of the structure you can see the variables that may already sound familiar from the module reader. These parameterize the memory allocation and the input source for the module. The input functions are usually set so that the characters come from the module reader, but there is no principal objection to use other character source for the purpose.

The variable `pvInput` is not altered by the module. It is only passed to the input functions. The function pointer name `pfGetCharacter` speaks for itself. It is like `getc` returns the next character. However when this function pointer is set to point to the function `reader_NextCharacter` the input is already preprocessed a bit. Namely the `include` and `import` directives were processed.

This imposes some interesting feature that you may recognize now if you read the reader module and this module definition carefully. `include` and `import` works inside multi-line strings. (OK I did not talk about multi-line strings so far so do not feel ashamed if you did not realize this.)

The function pointers `pfFileName` and `pfLineNumber` should point to functions that return the file name and the line number of the last read character. This is something that a `getc` will not provide, but the reader functions do. This will allow the lexical analyzer to store the file name and the line number for each token.

The next group of variables seems to be frightening and unreadable at first, but here is this book to explain them. These variables define what is a string, a symbol, what has to be treated as unimportant space and so on. Usually symbols start with alpha character and are continued with alphanumeric characters in most programming languages. But what is an alpha character? Is `_` one or is `$` a valid alphanumeric character. Well, for the lexer module if any of these characters appear in the variable `SSC` then the answer is yes. The name stands for *Symbol Start Characters*. But lets go through all these variables one by one.

`char *SSC;`

This *Symbol Start Character* variable contains all the characters that may be used to start a symbol. This symbol can be a variable or a symbol that appears for itself in the code like in the command `SET FILE`. (See the users guide.)

```
QWERTZUIOPASDFGHJKLYXCVBNMqwertzuiopasdfghjklyxcvbnm_:$
```

`char *SCC;`

This *Symbol Continuation Character* variable contains all the characters that may be used inside a symbol after the opening first character. The default value for this variable is

```
QWERTZUIOPASDFGHJKLYXCVBNMqwertzuiopasdfghjklyxcvbnm_1234567890:$
```

`char *SFC;`

This *Symbol Finishing Character* variable contains all the characters that may be used as the last character inside a symbol. The default value for this variable is

```
QWERTZUIOPASDFGHJKLYXCVBNMqwertzuiopasdfghjklyxcvbnm_1234567890$
```

which works fine for ScriptBasic. Note that this prohibits ScriptBasic variables to finish with colon.

`char *SStC;`

This *Start String Character* variable contains the characters that may start a string. The ScriptBasic value contains only the `"` character thus ScriptBasic strings can only start and end with the `"` character. However some other languages may use different string starting and finishing characters.

If there are more than one characters in this string then a string opened using a character should be closed using the same character. This is hard coded into the C program of the lexer.

The lexer also recognizes single-line strings and multi-line strings. A single-line string starts with a single `"` (or whatever characters are allowed in the `SStC` field) and finish with a single `"`. There can not be new-line character in a single-line string and any `"` character in the string should be quoted using the `\` character. The `\` character is not hard-coded it is configured in the field `ESCS`, as you will see later.

A multi-line string starts with `"""` characters that is three `"` characters and finishes the same way. Multi-line string may span several lines. This notation of multi-line string was inherited from the language Python. (At least I did not see it anywhere else.)

`char *SKIP;`

This *Skip* variable contains all characters that are to be skipped. This is the space, tab and the carriage-return character in case of ScriptBasic.

Skipping these characters does not mean that these characters are not taken into account. They serve a very important role: they stop tokens, thus no space can appear inside the name of a variable for example. However there is no token generated from these characters.

Note that the carriage-return character included in this string allows ScriptBasic to compile any DOS edited and binary transferred files under UNIX. However the operating system may have problem with the terminating carriage-return on the very first line.

`char *ESCS;`

This *Escape String* variable list all those characters that can be escaped in a string. The line that initializes this variable in `lex_InitStructure`:

```
pLex->ESCS = "\\n\nt\tr\r\"\"\'\'";
```

The first character of the `ESCS` string is the character used to escape other characters. This is the `\` character for ScriptBasic. The latter characters list the original character on the odd positions and the replacement characters on the following even position. For example the second character of this string is `n` and the replacement character is a new-line character, thus `\n` will be new-line in any sinle- or multi-line string in a BASIC program.

`long fFlag;`

This variable is a bit field that controls how numbers are treated in strings. The lines that initialize this variable are

```
pLex->fFlag = LEX_PROCESS_STRING_NUMBER       |
              LEX_PROCESS_STRING_OCTAL_NUMBER |
              LEX_PROCESS_STRING_HEX_NUMBER   |
              0;
```

The constants defined also in '`lexer.c`' tell the lexical analyzer that an escape character in a string followed by numeric characters should be converted to characters of the code. This the string `"a\10a"` will contains two `a` character separated by a new line. When the first character following the escape character is `0` the numbers are treated as octal numbers. If this character is `x` (lower case only and not `X`) the number is treated as hexadecimal. The escaped number is as long as there are numbers following each other without space. If the number is hexadecimal the letters `a-f` and `A-F` are also treated as digits.

The default values for these variables are set in the function `lex_InitStructure`. Interestingly these default values are perfectly ok for ScriptBasic.

The field `pNASymbols` points to an array that contains the non-alpha symbols list. Each element of this array contains a string that is the textual representation of the symbol and a code, which is the token code of the symbol. For example the table `NASYMBOLS` in file '`syntax.c`' is:

```
LexNASymbol NASYMBOLS[] = {
{ "
```

2.3.1 Functions implemented in this module

```
The following subsections list the functions that are implemented in this module. T
Documentation embedded in the C source as comment. Treat these subsections more as r
```

2.4 Syntax Analyzer

The syntax analyzer is a module that reads the token stream delivered by the lexer m
ule and builds a memory data structure containing the syntactically an-
alyzed and built program. The syntax analyzer is contained in the source file ‘expr
ule come from the fact that the most important and most complex task of syn-▮
tax analysis is the analysis of the expressions.

For the syntax analyzer the program is a series of commands. A command is a se-▮
ries of symbols. There is nothing like command blocks, or one command em-▮
bedding another command. Therefore the syntax definition is quite simple and yet st
erful enough to define a BASIC like language.

Because syntax analysis is quite a complex task and the syntax analyzer built for S
Basic is quite a complex one I recommend that you first read the tutorial from the S
Basic web site that talks about the syntax analysis. This is a series of slides to-
gether with real audio voice explaining the structure of the syntax an-
alyzer of ScriptBasic.

The syntax analyzer should be configured using a structure containing the con-▮
figuration parameters and the global variables for the syntactical ana-
lyzer. This structure contains the pointer to the array containing the syn-▮
tax definition. Each element of the array defines command syntax. Command syn-▮
tax is the list of the symbols that construct the command. When the syn-
tactical analyzer tries to analyze a line it tries the array elements un-▮
til it finds one matching the line. When checking a line against a syn-
tax definition the syntactical analyzer takes the lexical elements on the line and
bol in the syntax definition. A symbol can be as simple as a reserved word, like if
tax element is matched by the specific keyword. On the other hand a sym-
bol on the syntax definition can be as complex as an expression, which is matched b

The syntax analyzer has some built in assumption about the language, but the ac-▮
tual syntax is defined in tables. This way it is possible to analyze dif-▮
ferent languages using different tables in the same program even in the same pro-▮
cess in separate threads.

When the syntax analyzer reads the syntax definition of a line and matches the to-
kens from the lexer against the syntax element it may do several things:

- recognizes that the syntax element matches the coming token and goes on
- recognizes that the syntax element matches the coming token(s) and cre-
  ates one or more new nodes in memory that hold the values associated with the to
- recognizes the syntax element, does not match it against any token, and does so

The first is the case when the syntax element is a constant symbol. For ex-
ample it is a command keyword. In this case there is nothing to do with the key-
word except that the syntax analyzer has to recognize that this is the state-
ment identified by the keyword. The actual code will be generated later when non-
constant syntactical elements are found.

When the syntax analyzer sees that the next syntax element is some vari-
able, non-constant syntax element it matches the coming tokens and cre-
ates the nodes that hold the actual value for the tokens. For example when the syn-
tax element is string the syntax analyzer checks that the coming token is a string
ates a node that holds the string. The most important example is the syn-
tax element expression. In this case the syntax analyzer checks that the com-
ing tokens form an expression and not only "consumes" these tokens, but cre-
ates several nodes that hold the structure of the expression.

We can distinguish between constant and variable symbolic definition elements.

- A constant symbolic element is matched by a constant symbol. A con-
  stant symbolic element is a reserved keyword, or a special character that should
  pear at a certain position on the line.
- A variable symbolic element on the other hand is matched by several dif-
  ferent actual values. The simplest example of a variable symbolic el-
  ement is a number. A number in the syntax definition tells the ana-
  lyzer that a number should appear at the position on the line. How-
  ever it does not specify the value of the number. Any number can ap-
  pear and is valid at the position. When a variable symbolic element is matched t
  tual value, which was presented on the line and matched the symbolic def-
  inition element is stored in the memory structure that the analyzer builds.

There are some special symbols that are always matched whenever they are checked by
tax analyzer. They do not consume any lexical element from the line, and gen-
erate values in the memory structure that the analyzer builds.

The symbolic definition elements are:

expression This element matches an expression. When this syntax def-
inition element should be matched the syntactical analyzer tries to work up an e
pression starting from the actual position in the lexical unit stream.

expression_list This element matches a list of expressions separated by comma c

string This element is matched by a single string. Whenever you think to use thi
tax definition element consider using expression instead. This ele-
ment matches only a single string and not an expression resulting string value.

integer This element is matched by an integer number. Whenever you think to use
tax definition element consider using expression instead. This ele-
ment matches only a single string and not an expression resulting in-
teger value.

float This element is matched by an integer or float number. Whenever you think
tax definition element consider using expression instead. This ele-
ment matches only a single string and not an expression resulting in-
teger or float value. Implementing a command that requires this sym-
bolic element should accept integer values at the same location be-
cause this value matches any float or integer value. Any integer value passed o
tax location is converted to float during compile time. (Note that the C code d
bers are stored and handled using the C type double.)

symbol This element accepts a symbol. Before using this syntax def-
inition element you should be familiar with the other elements that may ac-
cept a symbol for a certain role. You should use this element when want-
ing to deal with the actual name of the symbol during run time. Note how-
ever that there are other elements that you should consider before us-
ing this syntax definition element.

absolute_symbol This element accepts a symbol similar to the syntax def-
inition element symbol. The difference is that symbol accepts a rel-
ative symbol which is treated as belonging to the current name space un-
less explicit name space was defined. Absolute symbol is taken with-
out any modification.

name_space This syntax definition element is matched by an absolute sym-
bol and sets the current name space.

end_name_space this syntax definition element does not consume any lex-
ical elements, but closes a name space and the surrounding name space is used a

lval This element can be matched by a left value. That is some vari-
able reference to which value can be assigned.

lval_list This element can be matched by a list of left values. A list is sev-
eral left values separated by commas.

local_start This syntax definition element is always matched, and does not con-
sume any lexical element from the list. When this syntax definition el-
ement is reached by the syntax analyzer it starts a new local scope. Such a poi
ally the start of a function or procedure.

local_end This syntax definition element is the pair of local start. This syn-▮ tax definition element is always matched, and does not consume any lex-▮ ical element from the list. When this syntax definition element is reached by th tax analyzer it finishes the local scope. Such a point is usually the end of a i tion or procedure. Note that local scopes can not be nested.

local This syntax definition element is matched by a symbol, which is treated as cal variable. The symbol is modified according to name space. The syn-▮ tax definition element is not matched whenever it is tried to be used out-▮ side of local scope.

local_list This symbol definition element is matched by a comma sep- arated list of local.

function This symbol definition element is matched by a symbol, which is treated tion name. Note that the local scope does not automatically start when such a sy tax definition element is matched.

thisfn This symbol definition element is matched within a local scope by the nam tual function or procedure. This is usually used to describe the as- signment that assigns a return value to the function name.

label This symbol definition matches a label, which is usually used af-▮ ter goto like instructions.

label_def This symbol definition is matched by a symbol. This symbol is go-▮ ing to be treated as a label. All labels are global.

go_back

go_forward

come_back

come_forward These symbol definition elements should be used to de- fine block and looping structures. Whenever an instruction like for/next or if/t fine where to continue execution based on the condition. The symbol def-▮ inition elements go-forward and come-back place the current instruc- tion location on a compile time stack. The symbol definition elements go-▮ back and come-forward take the last element from the same stack. They also check cation was placed on the stack by a matching construct, assuring that no out of der nesting structures appear, like if/for/endif/next.

Note that you can find other syntax definition elements in the file syn- tax.def. However these are converted to a character value by the Perl script tool sy taxer.pl These pseudo syntax definition elements are:

nl end of line ('\n' character)

tab tab character ('\t' character)

2.4.1 Name Space

The scriba syntax analyzer implements a simple name space handling. A vari-▮ able, label or function name always belongs to a name space. The default name space tax analyzer processes a non-absolute symbol it converts the name to contain the name space. A variable named var in the name space main has the name main: tax analyzer automatically converts the name to contain the name space, therefore main::var and var are equivalent when used in the name space main.

When a variable contains double colon it is treated as an absolute name, and no nam ule and use the variable name main::var it will NOT be converted to module::main::v son is that it already contains the characters :: and therefore scriba assumes that it already contains the name space.

If you are in the name space module and want to refer to the variable module::main: mat contains all nested name spaces to the variable. The second version tells the s tax analyzer that the variable is relative, altough it contains double colon. This cause it starts with double colons.

If you are in name space module::submodul and want to refer to the variable module::var you can either write module::var or _::var. The first format contains all nested name spaces to the variable. The second version tells the s tax analyzer that the variable is relative and the base module is the embedding module of the current one.

If you are familiar with the UNIX or DOS/Windows directory notation you can find si ilarities of file names and name space in scriba. In file names double dot means th ent directory. In scriba underscore character means the parent name space. You can character not only in front of a name, but also within :: characters. For example

```
Main::module::var
Main::module::submodule::_::var
Main::_::Main::module::var
```

are equivalent.

Name spaces help to separate variables and to develop scripts cooperatively, but do hibit one name space to access variables or symbols of other name spaces.

2.4.2 Expression

The formal description of an expression syntax is:

```
tag ::= UNOP tag
        NUMBER
        STRING
        '(' expression ')'
        VARIABLE  '[' expression_list ']'
```

```
                 FUNC '(' expression_list ')'
                 .

      expression_list ::= expression [ ',' expression_list ] .
      expression_i(1) ::= tag .
      expression_i(i) := expression_i(i-1) [ OP(i) expression_i(i) ] .
      expression ::= expression_i(MAX_PREC) .
```

where

OP(i)   is a binary operator having precdence I

UNOP is an unary operator

NUMBER is a number (float or integer)

STRING is a string

VARIABLE is a symbol that is to a local or global variable

FUNC is a function name, which is either a built-in function or a user defined function

MAX_PREC is the maximal precedence

The syntax analyzer is written to match an expression whenever an expression syntax definition element is to be matched according to these rules. The list in function, unary operators and binary operators are defined in the module "global" variables, BuiltInFunctions, Unaries, and Binaries.

2.4.3 Functions implemented in this module

The following subsections list the functions that are implemented in this module. The source text of this documentation was extracted from the source Documentation embedded in the C source as comment. Treat these subsections more as erence documentation and less tutorial like.

2.5 Builder

The module is implemented in the file ‘builder.c’.

The rule of the builder is to compile the code created by the syntax an-
alyzer into a continuous memory area. This compilation phase has two ad-
vantages. First of all it results a compact code which can easily saved into ex-
ternal file and can also be loaded making recompilation unnecessary be-
fore each execution. The second advantage is that the resulting code is smaller and

When the syntax analyzer starts its work the final size of the code is not known. T
fore the syntax analyzer starts to build up a memory structure using point-
ers, linked lists allocating memory step by step as the code grows. Each com-
mand, expression element is stored in an internal structure, which is called a node
ample a node containing the operation "plus" contains the node type, which says it
erator "plus" and contains two pointers to the two operands. The operands are also
ber the node contains the value of the number and the node type telling that it is
ber. If the operand is a variable the node type tells that the node is a vari-
able and the node contains the serial number of the variable. If the operand needs
ther evaluation then the node is probably an operation having arguments pointed by

The structure that the builder creates is same as that of the syntax an-
alyzer but it is allocated in a single memory chunk and instead of point-
ers it uses indices to refer a node from another. These indices are num-
bered from 1 and not from zero. This is because the index zero is used for NULL poi

2.5.1 Node Structure

Have a look at the C definition of a node:

```
typedef struct _cNODE
  long OpCode; // the code of operation
  union
    struct // when the node is a command
      unsigned long next;
      union
        unsigned long pNode;// node id of the node
        long lLongValue;
        double dDoubleValue;
        unsigned long szStringValue;
        Argument;
      CommandArgument;
```

```
          struct //when the node is an operation
            unsigned long Argument;//node id of the node list head
            Arguments;
          union // when the node is a constant

            double dValue;
            long   lValue;
            unsigned long sValue; // serial value of the string from the string tabl
            Constant;
          struct // when the node is a variable
            unsigned long Serial;// the serial number of the variable
            Variable;
          struct // when node is a user functions
            unsigned long NodeId; // the entry point of the function
            unsigned long Argument; // node id of the node list head
            UserFunction;
          struct // when the node is a node list head
            unsigned long actualm; //car
            unsigned long rest;    //cdr
            NodeList;
          Parameter;
         cNODE,*pcNODE;
```

The field OpCode is the same as the code used in the lexer or the syntax an-
alyzer. In case of an IF statement it is CMD_IF. This field can, should and is used
tify which part of the union Parameter is to be used.

The individual lines of the BASIC program that create code are chained into a list.
LST. This type of node contains NodeList structure. The field NodeList.actualm con-
tains the index of the first node of the actual line and the field NodeList.rest co
tains the index of the next header node.

This type of node is used to gather expression lists into a linked list.

Note that usually not the first node in the byte-code is the first head node, where
erated from a line are created before the head node is allocated in the syn-
tax analyzer and the head node thus gets a larger serial number. The builder uses t
rial numbers counted by the syntax analyzer and does not rearrange the nodes.

The command node that the field NodeList.actualm "points" contains the op-
code of the command. For example if the actual command is IF then the OpCode is CMD
IF.

In case of command nodes the Parameter is CommandArgument. If the command has only
gle argument the field next is zero. Otherwise this field contains the node in-
dex of the node holding the next argument.

The Parameter.CommandArgument.Argument union contains the actual argument of the co
mand. There is no indication in the data structure what type the argument is. The c
mand has to know what kind of arguments it gets, and should not interpret the union

The field pNode is the node index of the parameter. This is the case for ex-
ample when the parameter is an expression or a label to jump to.

The fields lLongValue, dDoubleValue and szStringValue contain the constant val-
ues in case the argument is a constant. However this is actually not the string tha
dex to the string table where the string is started. (Yes, here is some in-
consistency in naming.)

Strings are stored in a string table where each string is stored one af-
ter the other. Each string is terminated with a zero character and each string is p
ceded by a long value that indicates the length of the string. The zero char-
acter termination eases the use of the string constants when they have to be passed
erating system avoiding the need to copy the strings in some cases.

The field Parameter.CommandArgument.next is zero in case there are no more ar-
guments of the command, or the index of the node containing the next ar-
gument. The OpCode field of the following arguments is eNTYPE_CRG.

When the node is part of an expression and represents an operation or the call of a
in function then the Arguments structure of the Parameter union is to be used. This
ply contains Argument that "points" to a list of "list" nodes that list the ar-
guments in a list. In this case the OpCode is the code of the built-in func-
tion or operation.

When the node represents a string or a numeric constant the Constant union field of
stant value similar as the field CommandArgument except that it can only be long, d
stant node the OpCode is eNTYPE_DBL for a double, eNTYPE_LNG for a long and eNTYPE
STR for a string.

When the node represents a variable the field Variable has to be used. In this case
tains the serial number of the variable. To distinguish between local and global va
ables the OpCode is either eNTYPE_LVR for local variables or eNTYPE_GVR for global

When the node is a user defined function call the field UserFunction is used. Note
erated from the line sub/function myfunc but rather when the function or sub-
routine is called. The OpCode is eNTYPE_FUN.

The field NodeId is the index of the node where the function or subrou-
tine starts. The field Argument is the index of the list node that starts the list
gument expressions.

2.5.2 Binary File Format

The built code is usually saved to a cache file and this file is used later to load
ready compiled code into memory for subsequent execution. The format of this file c
tion build_SaveCode in file 'builder.c'. This function is quite linear it just save
eral structures and is well commented so you should not have problem to un-█
derstand it. However here I also give some description on the format of the bi-█
nary file.

The binary file may or may not start with a textual line. This line is the usual UN
erating system how to execute the text file. Altough we are talking now about a bi-
nary file, from the operating system point of view this is just a file, like a Scri
Basic source file, a Perl script or a bash script. The operating system starts to r
thing like

```
#! /usr/bin/scriba\n
```

with a new-line character at the end then it can be executed from the com-█
mand line if some other permission related constraints are met.

When ScriptBasic saves the binary format file it uses the same executable path that
#! /usr/bin/mypath/scriba and the basic progam 'myprog.bas' was started us-█
ing the command line

```
/usr/bin/scriba -o myprog.bbf myprog.bas
```

then the file 'myprog.bbf' will start with the line #! /usr/bin/mypath/scriba.█

The user's guide lists a small BASIC program that reads and writes the bi-█
nary file and alters this line.

Having this line on the first place in the binary format BASIC file makes it pos-█
sible to deliver programs in compiled format. For example you may develop a CGI ap-
plication and deliver it as compiled format to protect your program from the cus-█
tomer. You can convert your source issuing the command line

```
/usr/bin/scriba -o outputdirectory/myprog.bas myprog.bas
```

and deliver the binary 'myprog.bas' to the customer. ScriptBasic does not care the
tension and does not expect a file with the extension .bas to be source BA-█
SIC. It automatically recognizes binary format BASIC programs and thus you need no
ter even the URLs that refer to CGI BASIC programs.

The next byte in the file following this optional opening line is the size of a lon
chine the code was created. The binary code is not necessarily portable from one ma

chine to another. It depends on pointer and long size as well as byte or-▮
dering. We experienced Windows NT and Linux to create the same binary file but this

The size of a long is stored in a single character as sizeof(long)+0x30 so the ASCI
acter is either '4' or '8' on 32 and 64 bit machines.

This byte is followed by the version information. This is a struct:

```
       unsigned long MagicCode;
       unsigned long VersionHigh, VersionLow;
       unsigned long MyVersionHigh,MyVersionLow;
       unsigned long Build;
       unsigned long Date;
       unsigned char Variation[9];
```

The MagicCode is 0x1A534142. On DOS based system this is the characters 'BAS' and ^
sue the command

```
     C:\> type mybinaryprogram.bbf
```

you will get

```
     4BAS
```

without scrambling your screen. If you use UNIX system then be clever enough not to
nary program to the terminal.

The values VersionHigh and VersionLow are the version number of Script-
Basic core code. This is currently 1 and 0. The fields MyVersionHigh and MyVersionL
served for developers who develop a variation of ScriptBasic. The vari-
ation may alter some features and still is based on the same version of the core co
sion fields are reserved here to distinguish between different variation ver-▮
sions based on the same core ScriptBasic code. To maintain these version num-▮
bers is essential for those who embed ScriptBasic into an application, es-▮
pecially if the different versions of the variations alter the binary file for-▮
mat which I doubt is really needed.

The field Build is the build of the core ScriptBasic code.

The Date is date when the file 'builder.c' was compiled. The date is stored in a lo
sures that no two days result the same long number. In case you want to track how t
tion build_MagicCode in file 'builder.c'. This is really tricky.

The final field is Variation which is and should be an exactly 8 charac-
ter long string and a zero character.

If you want to compile a different variation then alter the #define directives in the file ‘builder.c’

```
#define VERSION_HIGH 0x00000001
#define VERSION_LOW  0x00000000
#define MYVERSION_HIGH 0x00000000
#define MYVERSION_LOW  0x00000000
#define VARIATION "STANDARD"
```

To successfully load a binary format file to run in ScriptBasic the long size, the r
sion information including the build and the variation string should match. Date ma

The following foru long numbers in the binary file define

the number of global variables,

the number of the nodes of the compiled program

the index of the start node where the execution should be started

the length of the string table

This is followed by the nodes themselves and the stringtable.

This is the last point that has to exist in a binary format file of a BASIC program. The following bytes are optional and may not be present in the file.

The optional part contains the size of the function table defined on a long and the
tion table. After this the size of the global variable table is stored in a long an
able table.

The global variable and function symbol table are list of elements, each containing a long followed by the zero character terminated symbolic name. The long st
rial number of the variable or the entry point of the function (the node index where the function starts).

These two tables are not used by ScriptBasic by itself, ScriptBasic does not need a
bolic information to execute a BASIC program. Programmers embedding ScriptBasic however demanded access global variables by name and the ability to execute individual functions from a BASIC program. If this last part is missing from a binary format BASIC program you will not be able to use in an application that uses these features.

2.5.3 Functions implemented in this module

The following subsections list the functions that are implemented in this mod-
ule. The source text of this documentation was extracted from the source
Documentation embedded in the C source as comment. Treat these subsections more as
erence documentation and less tutorial like.

2.6 Executor

The executor kills the code. Oh, no! I was just kidding. It executes the code, whic
thing different. Starts with the first node and goes on.

The execution of the code starts calling the function execute_Execute im-
plemented in the file 'execute.c'

This function initializes the execution environment that was not initial-
ized before calling the function execute_InitStructure

It allocates global variables, it fills command and instruction parame-
ters and finalizer function pointers with NULL and starts the function execute_
Execute_r

2.6.1 Command parameters

Each command type has a pointer that it can use for its own purpose. This is to avo
ing global variables. The file commands for example use the pointer avail-
able for the command OPEN. To access this pointer the macro PARAMPTR is used de-
fined in the file 'command.c'

2.6.2 Instrunction parameters

Not only the commands have a pointer for their use, but there are pointers available for each instruction. To make it clear:

If there are three OPEN statements in a program they share a *common command pointer*, but *each have its own instruction pointer*.

The code fragments implementing the different commands are free to use their own or
lated command or instruction pointer.

2.6.3 Finalizer function

Finalizer function pointers are available for each command type. This way they are
ilar to command parameters. There can be many OPEN statements in a program they share a common finalizer pointer. Each finalizer pointer is initialized to NULL.

The code fragments may put a function entry address in the finalizer pointer. When
ecution of a program is finished the executing function calls each function that has a non NULL pointer in the finalizer array.

The _r in the function name tells that this is a recursive function that may call i
self when an expression evaluation performs a function or subroutine call.

2.6.4 Functions implemented in this module

The following subsections list the functions that are implemented in this module. The source text of this documentation was extracted from the source Documentation embedded in the C source as comment. Treat these subsections more as
erence documentation and less tutorial like.

2.7 Configuration File Handling

ScriptBasic contains a fairly sophisticated configuration handling module. The configuration information is read each time the interpreter starts, there-fore it is vital that the information can be processed fast even if the configuration data is complex. This can be the case because the configuration information may also contain data for external modules that the interpreter loads when it starts and the external modules can access it any time they

The configuration information for ScriptBasic has to be maintained in textual format in a file that has more or less LISP syntax.

The format of the text file version of the configuration information is simple. It contains the keys and the corresponding values separated by one or more spa ally a key and the assigned value is written on a line. Lines starting with the cha acter ; is comment.

The values can be integer numbers, real numbers, strings and sub-configurations. St ther be single line or multi-line strings starting and ending with three """ characters, just like in the language ScriptBasic or in the language Python.

Sub-configurations start with the character (and are closed with the character). The list between the parentheses are keys and corresponding values.

This text file has to be converted to binary format. The ScriptBasic interpreter loads this binary format into memory without processing its content, thus loading speed of the configuration information is limited only by IO.

When the interpreter or an external module needs some configuration information there are functions in this module that can search and read information from the configuration file.

2.7.1 Functions implemented in this module

The following subsections list the functions that are implemented in this module. The source text of this documentation was extracted from the source Documentation embedded in the C source as comment. Treat these subsections more as erence documentation and less tutorial like.

2.8 Memory Allocation

This module is a general purpose memory allocation module, which can be used in any
that needs heavy and sophisticated memory allocation. Originally the mod-
ule was developed for
the ScriptBasic project. Later we used it for Index.hu Rt AdEgine project and multi
thread
features were introduced.

The major problem with memory allocation is that memory should be released. Old pro
grams depend
on the operating system to release the memory when the process exists and do not re
lease the memory
before program termination. Such programs are extremely difficult to port to multi-
thread operation.
In multi thread operation a thread my exit, but the memory still belongs to the pro
cess that goes on.

This module provides a bit of abstraction that helps the programmer to re-
lease the memory. The abstraction is
the following:

A piece of memory is always allocated from a segment. A segment is log-
ical entity and you should not think of
a segment in this content as a continuous memory area. I could also say that: when-
ever a piece of
memory is allocated it is assigned to a segment. When a piece of memory is re-
leased it is removed from the segment.
A segment is an administrative entity that keep track of the memory pieces that wer
located and assigned to the
segment.

To explain segment to the fines details: segments are implemented as linked lists.
ement of the list contains
the allocated memory piece as well as a pointer to the next and previous list membe

Whenever the programmer starts a sophisticated task that allocates sev-
eral memory pieces it has to create a new segment

and allocate the memory from that segment. When the memory is to be re-
lease the programmer can just say: release all the
memory from the segment. This way he or she does not need keep track of the al-
located memory structures, and walk through
the memory pointers of his or her program which are designed to the pro-
gram function instead of releasing the memory.

The overhead is the space allocated by two pointers for each memory piece and the s
ers for
each segment.

2.8.1 Functions implemented in this module

The following subsections list the functions that are implemented in this mod-
ule. The source text of this documentation was extracted from the source
Documentation embedded in the C source as comment. Treat these subsections more as
erence documentation and less tutorial like.

2.9 Variable Allocation

This module implemented in the source file 'memory.c' provides functions to al-
locate and deallocate memory for BASIC variables. This module itself al-
locates memory calling the underlying allocation module implemented in the file 'my
ule is to help ScriptBasic to reuse the allocated memory used for BASIC vari-
able value store effectively.

2.9.1 Functions implemented in this module

The following subsections list the functions that are implemented in this mod-█
ule. The source text of this documentation was extracted from the source
Documentation embedded in the C source as comment. Treat these subsections more as r
erence documentation and less tutorial like.

2.10 Error Reporting

2.10.1 Functions implemented in this module

The following subsections list the functions that are implemented in this mod-█
ule. The source text of this documentation was extracted from the source
Documentation embedded in the C source as comment. Treat these subsections more as r
erence documentation and less tutorial like.

2.11 The Logger Module

The logger module is included in ScriptBasic though it is not a critical part of it
mand line

```
version itself does not use the functions implemented in this module, though the fu
tions are
available for external modules. The Eszter SB Application Engine uses this mod-
ule to asynchronously
log hits and other events to ASCII text log files.
```

2.12 Hook Functions

2.13 Handle Pointers in External Modules Support Functions

2.14 Thread Support Functions

2.15 Dynamic Library Handling Support Functions

2.16 Other System Dependant Functions

2.17 Module Management

2.18 Run Time Options Handling

2.19 Simple Pattern Matching

2.20 Symbol Table Handling

The functions in this module implement a general purpose symbol table handling.

Generally a symbol table is a binding functionality that associates sym-
bols with
attributes. Symbols in this implementation is a zero terminated string, and the
attribute is a void * pointer. This is a general approach that can be used to store
trieve any kind of symbols.

The symbol table handling functions usually always return a void ** that can be mod
ified to point to the actual structure storing the attributes of the symbol.

The internal structure of a symbol table is a hash table of PRIME elements (211). E
nary table sorting the symbols.

3 Embedding the Interpreter

ScriptBasic was designed from the very start to be embeddable. This means that C pr
grammers having their own application can fairly easy compile and link Script-
Basic together with their application and have ScriptBasic as a built in script-
ing language in their application.

To do this the C programmer should use the C api implemented in the file ‘scriba.c’
ter we detail the C API as a reference listing all callable function, but be-
fore that there are some sections that describe a bit the overall model of Script-
BasiC. The next section will talk about what object orientation means for Script-
Basic and how to follow this object oriented approach when programming a Script-
Basic extended application in C.

3.1 Object Oriented Model of ScriptBasic

Although ScriptBasic is implemented in pure C the coding and developing con-
cept is rather object oriented. Such is the concept of the C API calling in-
terface. This means that you have to deal with an abstract ScriptBasic pro-
gram object when you want to execute a program. The structure of this pro-
gram object is totally private to ScriptBasic and as a programmer embed-
ding the interpreter you need not worry about it. The only action you have to do is
ate such an object before doing any other function call calling See ⟨un-
defined⟩ [scriba_new()], page ⟨undefined⟩ and to destroy it after the BA-
SIC program was executed and is not going to be used any more calling the func-
tion See ⟨undefined⟩ [scriba_destroy()], page ⟨undefined⟩.

The object is stored in memory and this piece of memory is allocated by Script-
Basic. The function See ⟨undefined⟩ [scriba_new()], page ⟨undefined⟩ al-
locates this memory and returns a pointer to this "object". Later this pointer has
fer to this object.

Because C is not object oriented the functions called should explicitly get this po
gument. When programming C++ the class pointer is used to access the class meth-
ods, and that also implicitly passes the object pointer to the method. The pointer
ing code is generated by the C++ compiler. When calling ScriptBasic API the pro-
grammer has to store the "object" pointer and pass it as first argument to any func

The type of the object pointer is pSbProgram.

3.2 Sample Embedding

The best way of learning is learning by example. Therefore here we will dis-█
cuss the most obvious embedding application: the command line variation of Script-█
Basic. The command line variation of ScriptBasic can be found in the file ‘scribacm
rectory ‘variations/standard’. You may also find there a file named ‘basiccmd.c’ th
tains the code that was used before the scriba_ C API was introduced. Have a look a

In this section we will present the code from the file, but for brevity some code w
gram develops the actual code may change while the one copied here most prob-█
ably remains the same. (The API definitions are not "hand" copied, but rather taken
umentation is compiled, so whenever the API changes the new documentation re-█
compiled reflects the change.)

3.2.1 Include Header Files

The main program of the standalone variation is implemented in the file ‘scribacmd.
gram you can see that it start including the file ‘scriba.h’. This file con-█
tains all definitions that are needed by the C compiler to compile the code call-█
ing the scriba_ API functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../../getopt.h"
#include "../../scriba.h"

#include "../../basext.h"
```

In case you miss the file ‘scriba.h’ then you should generate it using the pro-█
gram ‘headerer.pl’ or ‘headerer.bas’ which are also part of the distri-
bution. C header files in ScriptBasic are not maintained by themselves. The text is
tained inside the C code, and is extracted using one of these scripts. (They do the
son for this is to eliminate the double maintenance of function prototypes in the C

The file ‘basext.h’ is also #included by the main program. This is not usu-█
ally needed by other main programs. The standalone version needs it for the sole re
son to be able to print out on the usage screen the extension module in-
terface version that the actual interpreter support. This version is de-

fined in the macro INTERFACE_VERSION. Other than that there is no reason or need to
clude any other file than ‘scriba.h’.

3.2.2 Function main(), Variable Declarations

After some macro definitions that I do not list here the start of the function main comes:

```
main(int argc, char *argv[], char *env[]){

. . . .  variable declarations . . . . .


  pSbProgram pProgram;


. . . .  variable declarations . . . . .
```

This piece of code defines the local variables. The variable pProgram will hold the
gram object or simply saying pProgram is the program object variable. The other var
ables are not described here, their names and the code where they are used should m
age and purpose clear.

3.2.3 Command Line handling

The next piece of code is the command line option handling. This is not described here, because this is nothing special to ScriptBasic.

3.2.4 Creating Program Object

The real interesting piece of code starts at this fragment:

```
  pProgram = scriba_new(malloc,free);
```

This code first of all creates a new program object calling See ⟨undefined⟩
[scriba_new()], page ⟨undefined⟩. The two arguments to this function should be two p
ers to two functions that will be used by the interpreter to allocate and re-
lease memory. The two functions should exhibit behavior like malloc and free. All l
ers inherit these functions and call these functions to get and to release mem-
ory. In case there are more than one program objects used they may one af-
ter the other or in different threads at the same time use the same func-
tions or different functions. The only requirement is that the functions pointed by
guments should be thread safe if different threads are started executing Script-
Basic interpreters. Thread safeness also may play important role when some ex-
tension module is multi-thread.

3.2.5 Loading Configuration

```
scriba_LoadConfiguration(pProgram,pszForcedConfigurationFileName);
```

The next function call is See ⟨undefined⟩ [scriba_LoadConfiguration()],
page ⟨undefined⟩. This function gets the configuration information from the com-
piled configuration file. The details of configuration files is detailed in section
See ⟨undefined⟩ [Configuration File Handling], page ⟨undefined⟩.

It is not a must to call this function from the embedding application, but with som
ceptions most embedding applications will do. Most ScriptBasic interpreter pro-
gram objects need configuration information to be executed successfully.

To get the configuration for a program object there are two ways:

Load the configuration from a file.

Inherit the configuration information from another program object.

The second approach is necessary to gain performance in case there are sev-
eral interpreters running one after the other or in parallel threads in the same pr
cess. If all these interpreters need the same configuration information they can us
ory data, because the configuration information is not alterable by the in-
terpreter. This way you can save the successive file loading and the ex-
tra memory space.

When the configuration is inherited it is very important that the program ob-
ject holding the configuration is not destroyed while any of the inher-
iting program objects are alive. Therefore such applications usually cre-
ate a program object that does not run any program but holds the config-

uration information loaded calling See ⟨undefined⟩ [scriba_LoadConfiguration()], page ⟨undefined⟩ and that exists for the life of the whole process.

For more information how to inherit the configuration information see the function See ⟨undefined⟩ [scriba_InheritConfiguration()], page ⟨undefined⟩.

3.2.6 Loading Internal Preprocessors

The next function call loads the internal preprocessors. This is not necessary needed for the applications. This is needed only if there is some way to loa
ternal preprocessor before the program code is loaded. In the case of the standalone variation the command line option '-i' can specify internal preprocessor names.

```
       iError = scriba_LoadInternalPreprocessor(pProgram,pszIPreproc);
       if( iError )
         report_report(stderr,"",0,iError,REPORT_ERROR,&iErrorCounter,NULL,&fErrorF
         ERREXIT;
```

If this function is not called by the main program for the program object the interpreter will still process internal preprocessors that are referenced by the program code using the statement use.

3.2.7 Setting the File Name

To successfully load a program there is a need for the name of the file that holds
ify this file name for a program object the function See ⟨undefined⟩ [scriba_SetFile
page ⟨undefined⟩ should be used.

```
       scriba_SetFileName(pProgram,szInputFile);
```

This is the usual way getting the program into the program object, but there is another way. Some applications do not store the BASIC program string in text files, b
dia, like relation database. In that case the application has to load the program string into the application memory and call See ⟨undefined⟩ [scriba_LoadProgram
page ⟨undefined⟩ to compile and execute the BASIC program. However in this case the
gram should not contain any include or import statement.

3.2.8 Using the Cache

ScriptBasic is powerful being able to store the compiled version of BA-
SIC programs in binary format in file. The next piece of code checks if there is al
ready a compiled version of the BASIC program in the configured cache di-
rectory calling the function See ⟨undefined⟩ [scriba_UseCacheFile()], page ⟨un-
defined⟩

```
    if( scriba_UseCacheFile(pProgram) == SCRIBA_ERROR_SUCCESS ){
      if( (iError = scriba_LoadBinaryProgram(pProgram)) != 0 ){
        ERREXIT;
        }
```

The function See ⟨undefined⟩ [scriba_LoadBinaryProgram()], page ⟨undefined⟩ loads th
nary program.

3.2.9 Run External Preprocessors

If the cache file is not usable then the source text has to be loaded and com-
piled. Before that the external preprocessors has to do their jobs if there is any.
tion See ⟨undefined⟩ [scriba_RunExternalPreprocessor()], page ⟨undefined⟩ is called.

```
      iError=scriba_RunExternalPreprocessor(pProgram,pszEPreproc);
```

It is not a must for an application to call this function. Some applica-
tion may require the user to write their program in pure BASIC and not to use any p

3.2.10 Loading the Source Program

When the external preprocessors are done the program has to be loaded call-
ing the function See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨un-
defined⟩.

```
      if( scriba_LoadSourceProgram(pProgram) )ERREXIT;
```

This call loads the program source and processes it up to the point of ex-
ecution. All interpreter processing, file reading, inclusion of other files, lex-
ical, syntactical analysis and program building is done calling this function.

3.2.11 Saving Binary File

Before executing the loaded program the application may save the compiled binary version of the program. This can be done calling the function See ⟨undefined⟩ [scriba_SaveCode()], page ⟨undefined⟩ to save the binary code into a specific file, or calling the function See ⟨undefined⟩ [scriba_SaveCacheFile()], page ⟨undefined⟩ to save the binary code into the cache with an automatically calculated name.

```
         if( szOutputFile )
           if( isCoutput )
             scriba_SaveCCode(pProgram,szOutputFile);
           else
             scriba_SaveCode(pProgram,szOutputFile);
           if( !execute )exit(0);

          if( ! nocache )scriba_SaveCacheFile(pProgram);
```

3.2.12 Execute the BASIC Program

This is the step that all the previous steps were done. This function call to See ⟨undefined⟩ [scriba_Run()], page ⟨undefined⟩ executes the program.

```
       if( iError=scriba_Run(pProgram,CmdLinBuffer) )
      report_report(stderr,"",0,iError,REPORT_ERROR,&iErrorCounter,NULL,&fErrorFlag
         ERREXIT;
```

3.2.13 Destroying the Program Object

Although the program is executed and has finished still there is something extremely important that the application has to do: clean-up.

```
       scriba_destroy(pProgram);
```

The call to See ⟨undefined⟩ [scriba_destroy], page ⟨undefined⟩ releases all resources (memory) that were allocated to execute the BASIC program. Calling this function is not mandatory in a single process application that executes only one interpreter and exits from the process. In that case not calling this function the operating system is going to release the resources. However this is a bad practice.

3.3 ScriptBasic C API

The ScriptBasic C API is the high level programming interface that any programmer wanting to embed ScriptBasic into any application should use. This is the i
terface that the readily available variations of ScriptBasic use. This interface provides a general, high level abstraction interface to the ScriptBasic compilation and run-time system.

The API provides easy to use, and simple functions that can be used straightforward as well as more sophisticated, complex, harder to understand functions for special purpose application. The programmer can start with the simple functions and use the more complex ones as he or she developer the needs not fu
filled with the simpler functions.

The API delivers an object oriented calling interface even though this API is purel
ject oriented interface means that you have to call scriba_new to create a ScriptBasic object before any other call and you have to pass the returned pointer to eac
tion as first argument to operate on the actual BASIC program object.

This approach allow the programmer to allocate, load, compile, run, rerun, release multiple ScriptBasic programs even in multiple threads simultaneously with the limits that the operating system imposes not ScriptBasic.

4 Extension Modules

Extension modules are written usually using the language C and implement functions that can not be efficiently implemented in BASIC. These functions can be call
SIC programs just as if they were written in BASIC.

4.1 How Extension Modules are Used

To write external modules it is a good practice to learn first how ScriptBasic uses the modules.

External functions and external commands in ScriptBasic are declared using the declare sub or declare command statements. An example of such a statement is

```
     declare sub alma alias "trial" lib "ext_tial"
```

or

```
     declare command iff alias "iff" lib "ext_tial"
```

Following this declaration the function or the command can be used just as it were
plemented in BASIC.

```
     call alma(1,2,3)
```

The difference between external functions and external commands is the way ScriptBasic handles the arguments passed to them. Both external functions and external commands are implemented as C functions in the extension module compiled into a DLL or shareable object. Both of them look like user defined functions in the BASIC source code.

The difference is that external functions are called after the actual arguments are evaluated, while external commands are called without evaluating the arguments. Because of this external functions and external commands are implemented in C functions that have different prototypes. There is a pro
totype for external functions and a different one for external commands.

When ScriptBasic compiles this line the function or subroutine alma is defined just as a normal function or subroutine defined using the instructions of the basic language. Note that there are no differences other than syntax between subroutines and functions in ScriptBasic. When the program calls the fu
tion alma the ScriptBasic run-time system performs a function call to the basic function alma. In other words there is no difference from the caller point of v
tween the line above and the line:

```
Sub alma(a,b)
End sub
```

The function call can be performed in two different ways. One way is when the func-
tion appears in an expression. The other way is when the function is called us-
ing the call statement. There is no difference between the two calling pos-
sibilities from the internal operation point of view of the interpreter. This is be
cause CALL statement is implemented in a very simple way to simply eval-
uate the expression after the call statement and drop the result.

The calling code does not evaluate the expressions passed to the function as ar-
guments. This is usually the task of the functions. The functions get the node poin
pression list where the actual values for the arguments are and they can eval-
uate them.

The two different declarations declare sub and declare command differ in the way Sc
Basic interpreter handles the arguments. When an external function is de-
clared using the command declare sub the arguments are evaluated by the in-
terpreter before the function implemented in the external module is called. When an
ternal command is declared using the command declare command the arguments are *NOT*
uated by the ScriptBasic interpreter before calling the function imple-
mented in the external module. In the latter case the external function has to de-
cide if it wants certain arguments to be evaluated and can call the Script-
Basic function execute_Evaluate via the extension call-back table to eval-
uate the arguments. Also the prototype of a function declared using the state-
ment declare command is different from the prototype of a function declared us-
ing the command declare sub.

When a function is implemented externally ScriptBasic sees a declare sub state-
ment instead of a function or sub statement and starts to execute this state-
ment calling the code implemented in the file 'external.c' in the source di-
rectory 'commands'.

The name of the example function is alma as declared in the statement above. How-
ever this is only a symbolic name that exists only during syntax analy-
sis and is not available when the code is executed. The alias for the func-
tion is trial. This is the name of the function as it is implemented in the ex-
ternal module. When the interpreter executes this line the function name trial is u
tem call to locate the entry point. The module that contains the function is ext_
trial. The actual file name is 'ext_trial.dll' or 'ext_trial.so' or some other name
taining the given name and an extension specific to the operating system. Dur-
ing module load ScriptBasic automatically appends the appropriate exten-
sion as defined in the configuration file of ScriptBasic. (For further in-
formation on ScriptBasic configuration file syntax and location see the Script-
Basic Users' Guide!) ScriptBasic searches the module in the directories de-
fined in the configuration file and tries to load it using absolute file name. This
tem specific search paths are not taken into account.

When function implemented in an external module is first called the in-
terpreter checks if the module is loaded or not. If the module is not loaded the in
terpreter loads the module and calls module initialization functions. If the mod-
ule was already loaded it locates the entry point of the function and calls the fun

During module load ScriptBasic appends the appropriate dynamic load li-
brary extension and tries to load the module from the directories defined in the co
figuration file. It takes the directories in the order they are specified in the co
figuration file and in case it can load the module from a directory listed it stops

When the module is loaded ScriptBasic locates the function versmodu and calls it. T
tion is to negotiate the interface version between the external module and Script-
Basic. The current interface version is defined in the file ‘basext.c’ with the C m
VERSION. ScriptBasic calls this function to tell the module what version Script-
Basic supports. The function can decide if the module can work with the in-
dicated version and can answer: *yes it is OK, no it is not OK or yes, but I can sup
port only version X.* This is a negotiation process that finally result some agree-
ment or the module is abandoned if no agreement can be reached.

The function versmodu gets three arguments:

```
     int versmodu(int Version, char *pszVariation, void **ppModuleInternal)
```

The first argument is the version of the interface. The second argument is the ZCHA
minated 8-character string of the variation. This is STANDARD for the stan-
dard, stand alone, command line version of ScriptBasic. The ppModuleInternal pointe
ule pointer initialized to NULL. This pointer is hardly ever used in this func-
tion, but its address is passed as a third argument in case some appli-
cation needs it. The role of this pointer will be discussed later.

The function should check the parameters passed and return either zero in case it c
cept the interface or the highest interface it can handle. If this is the same as t
sion passed in the first argument the module should be accepted. If this is smaller
terface version offered by ScriptBasic the interpreter can decide if it can sup-
port the older interface required by the module.

If the function versmodu returns a version larger than the version offered the in-
terpreter will interpret this as a negotiation failure and will treat the mod-
ule as not loaded.

If there is no function named versmodu in the library ScriptBasic crosses the fin-
gers and hopes the best and assumes that the module will be able to work with the i
terface ScriptBasic offers. (should we change it to be configurable to dis-
allow such modules?)

After the successful version negotiation the interpreter calls the function named bootmodu. This function gets four arguments.

```
int bootmodu(pSupportTable pSt,
             void **ppModuleInternal,
             pFixSizeMemoryObject pParameters,
             pFixSizeMemoryObject *pReturnValue)
```

The first parameter is a pointer to the interface structure. This interface structure can and should be used to communicate with ScriptBasic. The second parameter points to the module pointer. The last two parameters are NULL for th
tion. The reason to pass two NULL pointers is that this is the prototype of each fu
tion callable by ScriptBasic implemented in the module. The last two parameters point to the parameters of the function and to the left value where the fu
tion value is to be returned. bootmodu actually does not get any parameter and should not pass any value back.

This function can be used to initialize the module, to allocate memory for the common storage if the functions implemented in the module keep some state information. If there is no function named bootmodu in the library file ScriptBasic assumes that the module does not need initialization. If the function bootmodu exists it should return zero indicating success or an error code. If an error code is returned the module is treated as failed. And an error is raised. (Errors can be captured using the BASIC ON ERROR GOTO command.

When this function returns the ScriptBasic interpreter evaluates the arguments and performs a call to the function named trial in our example.

When the program has finished the interpreter tries to locate the function finimodu in the module. This function may exist and should have the same prototype as any other function (except versmodu):

```
int finimodu(pSupportTable pSt,
             void **ppModuleInternal,
             pFixSizeMemoryObject pParameters,
             pFixSizeMemoryObject *pReturnValue)
```

This function can be used to perform clean-up tasks. The interpreter may call the f
tions of different modules in different threads asynchronously. (However it does no

Note that there is no need to release the allocated memory in case the module allocates memory using the memory allocation methods provided by the interface. Other resources may need release; for example files may need closing.

4.2 A Simple Sample Module

After you have got a slight overview how ScriptBasic handles the modules get a jump
start looking at the simplest sample module trial.c!

This module was the very first module the developers wrote to test the mod-
ule handling functionality of ScriptBasic. This does almost nothing, but prints out
sages so that you can see how the different functions are called in the mod-
ule. There is only one function in this module that the basic code can call. This i
tion increments a long value and returns the actual value of this state variable.

Here is the whole code:

```
#include <stdio.h>

#include "../../basext.h"

besVERSION_NEGOTIATE

  printf("The function bootmodu was started and the requested ver-
sion is %d\n",Version);
  printf("The variation is: %s\n",pszVariation);
  printf("We are returning accepted version %d\n",(int)INTERFACE_VERSION);

  return (int)INTERFACE_VERSION;

besEND

besSUB_START
  long *pL;

  besMODULEPOINTER = besALLOC(sizeof(long));
  if( besMODULEPOINTER == NULL )return 0;
  pL = (long *)besMODULEPOINTER;
  *pL = 0L;

  printf("The function bootmodu was started.\n");

besEND


besSUB_FINISH
  printf("The function finimodu was started.\n");
besEND

besFUNCTION(trial)
```

```
  long *pL;

  printf("Function trial was started...\n");
  pL = (long *)besMODULEPOINTER;
  (*pL)++;
  besRETURNVALUE = besNEWMORTALLONG;
  LONGVALUE(besRETURNVALUE) = *pL;

  printf("Module directory is %s\n",besCONFIG("module"));
  printf("dll extension is %s\n",besCONFIG("dll"));
  printf("include directory is %s\n",besCONFIG("include"));

besEND

besCOMMAND(iff)
  NODE nItem;
  VARIABLE Op1;
  long ConditionValue;


  USE_CALLER_MORTALS;

  /* evaluate the parameter */
  nItem = besPARAMETERLIST;
  if( ! nItem )
    RESULT = NULL;
    RETURN;

  Op1 = besEVALUATEEXPRESSION(CAR(nItem));
  ASSERTOKE;

  if( Op1 == NULL )ConditionValue = 0;
  else
    Op1 = besCONVERT2LONG(Op1);
    ConditionValue = LONGVALUE(Op1);


  if( ! ConditionValue )
    nItem = CDR(nItem);

  if( ! nItem )
    RESULT = NULL;
    RETURN;

  nItem = CDR(nItem);
```

```
    RESULT = besEVALUATEEXPRESSION(CAR(nItem));
    ASSERTOKE;

    RETURN;
  besEND_COMMAND
```

As you can see there is a lot of code hidden behind the macros. You can not see ver
cause they are implemented using the macro besVERSION_NEGOTIATE, besSUB_
START and bes_SUB_FINISH. These macros are provided in the header file ‘basext.h’,
initions and include statements that are needed to compile a module. To have a deep
derstanding feel free to have a look at the file ‘basext.c’ containing the source f

All the macros defined in the header file for the extensions start with the three l
ters bes. These stand for basic extension support.

The version negotiation function prototype and function start is created us-
ing the macro besVERSION_NEGOTIATE. This macro generates the function head with the
rameters named Version, pszVariation and ppModuleInternal. The accepted ver-
sion is returned using the macro INTERFACE_VERSION. The reason to use this macro is

The current version of the interface is 10 (or more). Later interfaces may prob-
ably support more callback interface functions, but it is unlikely that the in-
terfaces become incompatible on the source level. When the module is re-
compiled in an environment that uses a newer interface it will automat-
ically return the interface version that it really supports. If the in-
terfaces become incompatible in source level the compilation phase will most prob-
ably fail.

The bootmodu function is created using the macro besSUB_START. In this ex-
ample this allocates a state variable, which is a long. The memory allo-
cation is performed using a callback function and with the aid of the macro besALLO

Here we can stop a bit and examine, how the callback functions work. The Script-
Basic interpreter has several functions that are available for the exten-
sions. To access these functions the module should know the entry point (ad-
dress) of the functions. To get the entry points ScriptBasic creates a ta-
ble of the callback functions. A pointer to this table is passed as the first ar-
gument to each module function except the version negotiation function versmodu. In
tax this table is a struct named SupportTable.

There are numerous functions that an extension can and should use to com-
municate with ScriptBasic. One of the most important functions is memory al-
location. The field of the SupportTable named Alloc is initialized to point to the
tion alloc_Alloc defined in the file ‘myalloc.c’. This allocation func-
tion needs the size of the needed memory block and a pointer to a so-called mem-

ory segment. The memory segment pointer to be used is available via the SupportTabl
ber of the table is a pointer to another table containing the current in-
terpreter execution environment. This environment is also a &codestruct and con-
tains the memory segment pointer.

This is a bit complicated and you can get confused. To ease coding use the macros a
able. These will hide all these nifty details. However know that these macros as-
sume that you use them together. In other words besALLOC can only be used in a func
tion when the function head was created using the macro besFUNCTION (or besSUB_
START or besSUB_FINISH in case of bootmodu and finimodu). This is because the macro
sume certain variable names in the arguments.

The function finimodu is created using the macro besSUB_FINISH. This func-
tion in this example does nothing but prints a test message. There is no need to re
lease the memory allocated using besALLOC, because the memory is admin-
istered to belong to the segment of the interpreter execution environment and is re
leased by the interpreter before the exits.

The real function of the module is named trial and is defined using the macro besFU
able via the module pointer. The module pointer is always passed to the mod-
ule functions as the second argument. The functions can access it using the macro b
ing this macro it looks like a local variable.

To return the counter value the function needs a basic variable. This vari-
able should hold a long value and should be mortal. This is created us-
ing the macro besNEWMORTALLONG. The variable is actually a struct and the field con
taining the long value can be accessed using the macro LONGVALUE.

You can notice that this macro does not start with the letters bes. The rea-
son is that this macro comes from a different header file ‘command.h’. This header
cluded by ‘basext.h’ and the definitions in that file are mainly for im-
plementing internal commands linked statically. However some of the macros can be u
namic modules as well.

The function trial finally writes out some configuration data. On one hand this is
other example of a callback function used via a macro named besCONFIG. But this is
portant than that. This shows you that the modules can access any config-
uration data.

There is no need for any module to process separate configuration files. Script-
Basic reads the configuration file and stores each key and value pair it finds. It
thing for ScriptBasic itself, because they may be meaningful and needed by mod-
ules. The example module trial does not have its own data, therefore we print out t
figuration data that ScriptBasic surely has.

4.3 Compiling a Module

Compiling a module is easy and straightforward. Just do it as you would do for any
namic load library. On Linux you have to compile the source code to ob-
ject file, saying

```
     cc -c -o trial.o trial.c
     ld -d -warn-section-align -sort-comon -shared -o trial.so trial.o
```

assuming that the name of the file to compile is 'trial.c'. On other Unix op-
erating systems you have to issue similar commands.

On Windows NT you can use Visual C++ and create an empty project using the work spa
ard to create a new dll file. You have to add your source code to the project, se-
lect release version and compile the program.

Note that Windows NT dll files do not automatically export each non-static func-
tion of a program. They have to be declared in a DEF file or the functions should b
noted with a special type casting keyword. If you use the predefined macros avail-
able including the file 'basext.h' your functions will be exported with-
out creating a def file.

4.4 Installing a Module

A module is usually composed of two files. One file is the binary library mod-
ule with the extension .dll under Windows NT or .so under Unix. The other file is a
clude file, which contains the declare sub statement for each external function.

To install a module you have to copy or move the module binary to one of the di-
rectories specified in the configuration file as ScriptBasic module di-
rectory and you have to copy or move the include file into one of the di-
rectories specified in the configuration file as ScriptBasic include directory.

The basic program that uses the module includes the include file and it is ready to
tions declared in that file. Currently there are no install programs avail-
able that place the file at the appropriate locations.

4.5 Module Support Functions

Module support functions and macros are to ease the life of the extension programmers. They are defined in the file ‘basext.c’ and in the file ‘basext.h’ generated from ‘basext.c’ using the tool ‘headerer.pl’.

It is highly recommended that the extensions use these functions and that the extensions use the macros to call the functions the way the documentation suggests. The reason for this is to create readable code and to provide maintainabilit

5 Preprocessors

ScriptBasic is capable handling two kind of preprocessors. One is external preprocessors, the other one is internal preprocessor. The names *external* and the *internal* distinguish between the execution type of these preprocessors. External preprocessors are executed in a separate process. Internal preprocessors run in the interpreter thread.

Because of this external preprocessors are standalone command line tools, which may be written for any application and not specifically for ScriptBasic. You can edit the ScriptBasic configuration file so that you can use the C preprocessor, `m4` or `jamal` as a preprocessor. It is fairly easy to write an external preprocessor compared to internal preprocessors. External preprocessor reads a file and creates an output file. It need not know anything about the internal structures of ScriptBasic. Then only thing a ScriptBasic external preprocessor writer has to know is what it wants to do and how to read and write files.

Internal preprocessors are implemented in dynamic link libraries, work closely together with ScriptBasic and can not be started standalone. Internal preprocessors are written specifically for ScriptBasic and can and should access many of ScriptBasic internal structures.

From this you can see that external preprocessors are much easier to write, while internal preprocessors have much more possibilities. An internal preprocessor can never started as external and vice versa.

Before starting to write a preprocessor you have to carefully check what you want to gain and decide if you want to write an external preprocessor or an internal.

Because external preprocessors are just standalone programs and there is even a sample preprocessor HEB written in BASIC this chapter talks about the internal preprocessor capabilities. See Section 2.1 [External Preprocessor], page 6

5.1 Loading Preprocessor

Loading a preprocessor depends on the embedding application. To load an internal preprocessor the function `ipreproc_LoadInternalPreprocessor` is called (implemented in the file '`ipreproc.c`').

This function gets the name of an external preprocessor to load. The function searches the configuration information for the named preprocessor, loads the DLL/SO and invokes the initiation function of the preprocessor.

```
int ipreproc_LoadInternalPreprocessor(pPreprocObject pPre,
                                      char *pszPreprocessorName);
```

The first argument is the pointer to the ScriptBasic preprocessor object to access the configuration information and the list of loaded preprocessors to put the actual one on the list.

The second argument is the name of the preprocessor as named in the configuration file, for example

```
preproc (
  internal (
    sample "C:\\ScriptBasic\\bin\\samplepreprocessor.dll"
```

```
          )
```

The return value is zero or the error code.

(Note that this documentation may not be up-to date about the precise functioning of this function. For most up-to date information see the source documentation that is extracted from the source comment using the tool '`esd2html.pl`'.)

In the code base of ScriptBasic this function is called by the reader '`reader.c`' in the function `reader_LoadPreprocessors`. This is called automatically when the source is read and include files were also included. This function (`reader_LoadPreprocessors`) goes through all the lines and searches for lines that start with the word `preprocess` and name a preprocessor. The function loads the preprocessor and deletes the source line.

The function `ipreproc_LoadInternalPreprocessor` is also called from the function `scriba_LoadInternalPreprocessor` in source file '`scriba.c`'

This function can and should be used by the embedding programs to load all internal preprocessors that are to be loaded based on some external conditions. For example the VARIATION STANDARD of ScriptBasic (aka. the command line embedding variation) loads all internal preprocessors that were named after the command line option '`-i`'.

Other embedding application may get the information of desired preprocessors from different sources, like environment variables, configuration files and so on.

Note that internal preprocessors are not used whenever an already compiled version of the program is executing. If there is a `preprocess` line in the source code and the program has generated a cache or any other binary format BASIC program and that file is used to execute the program the preprocessor will no effect. The interpreter will not load the preprocessor and thus it will act as it did not exist.

There are two kind of preprocessors:

Altering preprocessors

The "altering" preprocessors act as conventional preprocessor altering the source code and/or altering the compilation environment during compilation that results finally a binary BASIC file. These preprocessors ask the ScriptBasic interpreter to unload the preprocessor before the actual execution of the program starts as they have nothing to do with the actual, executed program.

Debugger preprocessors

Debugger preprocessor collects symbolic information during the compilation phase and alters the hook functions `HOOK_ExecBefore`, `HOOK_ExecAfter`, `HOOK_ExecCall`, `HOOK_ExecReturn`.

During execution these "preprocessors" remain in the process and execute the functions that the altered hook pointers point to performing debugging, profiling or some other development support features.

The sample preprocessor `dbg` does this implementing a command line debugger.

Mixed preprocessors

It is possible, though I see no reason to write a preprocessor that belongs to both categories above.

Altering preprocessors are BASIC program specific and are usually invoked because the program contains a line `preprocess`.

Debugger type preprocessors are loaded the way of the execution of the program requests it. For example the user uses the option '`-i`'. In this case the command line version of ScriptBasic does not use cache file to ensure that the program really does using the preprocessor and starts the debugger, for example.

5.2 Skeleton of a Preprocessor

An internal preprocessor implemented as a `.dll` or `.so` file has to export a single function named `preproc`. The declaration of this function should look like this:

```
int DLL_EXPORT preproc(pPrepext pEXT,
                       long *pCmd,
                       void *p);
```

The first argument of the function is the preprocessor pointer. This pointer points to a structure. For each preprocessor loaded there is a separate structure of this type. This structure hold information on the preprocessor and there is some storage pointer in the structure that the preprocessor can access.

The definition of the structure is:

```
typedef struct _Prepext
  long lVersion;
  void *pPointer;
  void *pMemorySegment;
  struct _SupportTable *pST;
   Prepext, *pPrepext;
```

The field `lVersion` is the version of the interface that ScriptBasic wants to use when communicating with the preprocessor. If this version is not the same as the interface version that the preprocessor was compiled for then the preprocessor has to ask ScriptBasic not to use it and may return an error code or zero indicating that no error has happened. This may be useful in some cases when the preprocessor is optional and in case the preprocessor can not be loaded the program still may function. Read on how the preprocessor has to do this.

The field `pPointer` is initialized to `NULL` and is never changed by ScriptBasic. This is a pointer that can be used by the preprocessor to access its own thread local variables.

The field `pMemorySegment` is initialized to point to a memory segment that can be used via the memory allocation routines of ScriptBasic. These routines, however need not be linked to the DLL, because they are already in the process loaded as part of the executable and can be reached via the function support table.

This support table is pointed by the field `pST` and is the same type of `struct` as the support table available for the extension modules. Although this support table is the same type of `struct` it is not the same `struct`. The fields pointing to the different functions are eventually pointing to the same function, but when the program starts to execute a new support table is allocated and initialized. Thus there is no reason for the preprocessor to alter this table, because altering the table will have no effect on the execution of the program. Also the preprocessor if decides to stay in memory while the program is executed (for example a debugger), may rely on this support table even if a module altering the run-time support table is used.

In case there are more than one internal preprocessors used they will share the same support table. This way a preprocessor altering the preprocessor support table may alter the behavior of another internal preprocessor. However doing that need deep and detailed information of both ScriptBasic code and the other preprocessor and may result code that closely depends on the different versions of the different programs that work together.

The next argument to the function `pCmd` is an input and output variable. When the function is called by ScriptBasic it contains the command that the preprocessor is expected to perform. In other words this value defines the reason why the preprocessor was loaded. There are numerous points when ScriptBasic calls the preprocessor and at each point it sets this variable differently.

When the function returns it is supposed to set the variable `*pCmd` to one of the following values:

`PreprocessorContinue`

Returning this value means that the preprocessor has done what it was supposed to do and the program should go on.

`PreprocessorDone`

Returning this value means that the preprocessor has done what it was supposed to do and ScriptBasic should not, and indeed will not call any other preprocessors loaded for the actual "command".

`PreprocessorUnload`

Returning this value means that the preprocessor has done all tasks it had to do and ScriptBasic should close the preprocessor, release all memory that the preprocessor has allocated using the call-back function `alloc_Alloc` and should unload the dynamic load library.

The preprocessors function `preproc` may at any call return an `int` value. This value will be used by the interpreter as error code. This code is zero in case there was no error. This value has to be returned to ensure that the interpreter goes on. The error code `1` is used at any code in ScriptBasic to signal memory allocation problems. The symbolic constant `COMMAND_ERROR_PREPROCESSOR_ABORT` can be used to tell the interpreter to stop. For example the sample debugger preprocessor uses this error code when the user gives the command `q` to quit debugging.

5.3 Preprocessor Entry Points

Lets recall the prototype of the preprocessor function, which has to be implemented in each preprocessor:

```
int DLL_EXPORT preproc(pPrepext pEXT,
                       long *pCmd,
                       void *p);
```

This function has to be implemented in each internal preprocessor and is called when the preprocessor is loaded or when some the processing of the source program has reached a certain point. To inform the function about this point the argument `pCmd` is used. This argument points to a `long` that holds the a constant identifying the reason why the preprocessor function was called. The following subsections list these identifiers.

5.3.1 PreprocessorLoad

This entry point is used when the preprocessor is loaded. The pointer `p` is `NULL`.

When the preprocessor function is called with this argument it can be sure that this is the very first call to the function within the actual interpreter thread. It also can depend on the support function table and on the preprocessor memory segment pointer being initialized and ready to allocate memory.

It has to check that the version the preprocessor was designed and compiled for is appropriate and works together with the ScriptBasic interpreter that invoked the preprocessor. This can easily be done checking the version information in the preprocessor structure. Sample code:

```
if( pEXT->lVersion != IP_INTERFACE_VERSION )
  *pCmd = PreprocessorUnload;
  return 0;
```

This code is the appropriate place to allocate space for the preprocessor structure that hold the thread local variables. For example:

```
pDO = pEXT->pST->Alloc(sizeof(DebuggerObject),pEXT->pMemorySegment);
*pCmd = PreprocessorUnload;
if( pDO == NULL )return 1;
```

Note that this example is a simplified version of the one that you can find in the sample debugger preprocessor. This example uses the `Alloc` support function that is usually points to the function `alloc_Alloc` implemented in the file '`myalloc.c`'. When coding an external preprocessor you can rely on ScriptBasic that as soon as the preprocessor is unloaded the memory allocated using this function with the memory segment initiated for the preprocessor (like above) will be released.

The preprocessor has to return the error code or zero and may alter the value of the parameter `*pCmd` to `PreprocessorContinue` or `PreprocessorUnload`.

Although the calling code ignores the value returned in `*pCmd` unless it is `PreprocessorUnload` it is required by principle to set a value in this variable. The value `PreprocessorDone` can not be used when returning from this entry.

5.3.2 PreprocessorReadStart

This entry point is used before the source file reading starts. The pointer `p` points to the `ReadObject` used to read the source files. There is few uses of this entry point. You may alter the file handling function pointers or the memory allocation function pointers in the `ReadObject`.

This entry point is invoked only when the preprocessor load was initiated by external conditions, like command line option '`-i`' and never if the source code contained the preprocessor loading directive. This is because when the reader realizes that the preprocessor has to be loaded it is far over this point.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.3 PreprocessorReadDone0

This entry point is used when the reader has done the reading of the source file, but did not do any processing further. The parameter `p` points to the `ReadObject`. The preprocessor at this point can access the lines of the core BASIC program file without the included files and the very first line of the program that may be a Windows NT or UNIX special line (like `#!/usr/bin/scriba`) is still in the input lines.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.4 PreprocessorReadDone1

This entry point is used when the reader has done the reading the source file and unhooked the optional first Windows NT or UNIX start line (like `#!/usr/bin/scriba`), but did not process the included files. The preprocessor at this point can access the lines of the core BASIC program file without the included files. The parameter `p` points to the `ReadObject`.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.5 PreprocessorReadDone2

This entry point is used when the reader has done the reading the source file and unhooked the optional first Windows NT or UNIX start line (like `#!/usr/bin/scriba`), and has processed the included files. The preprocessor at this point can access the lines of the full BASIC program file with the included files. The parameter `p` points to the `ReadObject`.

This is the last point before the source code directive loaded preprocessors are invoked.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.6 PreprocessorReadDone3

This point is used when the reader has done all reading tasks, processed and linked the include files, has removed the first Windows NT or UNIX specific line, and loaded the preprocessors that were to be loaded by program directive. The parameter `p` points to the `ReadObject`.

At this point the preprocessor may check the source code and alter it according to its need. All processing should be done here that needs the characters of the source file. Careful decision has to be made whether using this point of entry to alter the source file or the entry point `PreprocessorLexDone` when the source file is already tokenized.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.7 PreprocessorLexInit

This entry point is used when the lexer object was allocated and initialized. The pointer `p` points to the `LexObject`.

The preprocessor at this point may alter the `LexObject` parameters. Here is a copy of the function `lex_InitStructure` from ScriptBasic v1.0build26. (other versions may slightly differ).

```
void lex_InitStructure(pLexObject pLex
  )
/*noverbatim
CUT*/
  pLex->pfGetCharacter = NULL;
  pLex->pfFileName = _MyFileName;
  pLex->pfLineNumber = _MyLineNumber;
  pLex->SSC = "QWERTZUIOPASDFGHJKLYXCVBNMqwertzuiopasdfghjklyxcvbnm_:$";
  pLex->SCC = "QWERTZUIOPASDFGHJKLYXCVBNMqwertzuiopasdfghjklyxcvbnm_1234567890:$";
  pLex->SFC = "QWERTZUIOPASDFGHJKLYXCVBNMqwertzuiopasdfghjklyxcvbnm_1234567890$";
  pLex->SStC = "\"";
  pLex->ESCS = "\\n\nt\tr\r\"\"\'\'";
  pLex->fFlag = LEX_PROCESS_STRING_NUMBER       |
                LEX_PROCESS_STRING_OCTAL_NUMBER |
                LEX_PROCESS_STRING_HEX_NUMBER   |
                0;
  pLex->SKIP = " \t\r"; /* spaces to skip
                           \r is included to ease compilation of DOS edited
                           binary transfered files to run on UNIX */
  pLex->pNASymbols = NULL;
  pLex->pASymbols  = NULL;
  pLex->pCSymbols  = NULL;
  pLex->cbNASymbolLength = 0; /* it is to be calculated */

  pLex->buffer = lexALLOC(BUFFERINCREASE*sizeof(char));

  if( pLex->buffer )
    pLex->cbBuffer = BUFFERINCREASE;
  else
    pLex->cbBuffer = 0;

  CALL_PREPROCESSOR(PreprocessorLexInit,pLex);
```

(Note `CALL_PREPROCESSOR` is a macro that call the preprocessor with appropriate arguments.)

The preprocessor may decide for example to alter the string `SSC` that contains all characters that may start a symbol, or `SCC` that contains all characters that can part a symbol or `SFC` that contains all characters that can be the final character of a symbol. This way for example a preprocessor may set these strings that allows Hungarian programmers to use

ISO-Latin-2 accented letters in their variables. (However those characters are going to be case sensitive.)

The preprocessor may also set the pointers that point to the tables that contains the alphanumeric symbols (`pASymbols`), non-alpha symbols (`pNASymbols`) and the table used for some ScriptBasic internal debugging purpose (`pCSymbols`).

The preprocessor may also release and reallocate a smaller or larger `buffer` if wishes. (I personally see no reason.)

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.8 PreprocessorLexDone

This entry point is used when the lexer has finished the lexical analysis and the list of tokens is already in the memory. The pointer `p` points to the `LexObject`.

At this point the preprocessor may alter the tokenized form of the BASIC program. The list of tokens still contains the comment lines (also tokenized although it may make no sense), and the continuation lines are still being split containing the `_` character and the new-line token.

The preprocessor may gain information from comments in case the some comments provide information for the preprocessor.

If the preprocessor uses some special symbols that drive the preprocessor processing but should be removed from the token list the preprocessor at this point may unlink the token from the list or just set the `type` of the token to `LEX_T_SKIP` or `LEX_T_SKIP_SYMBOL`.

If you do not or do not want to understand the difference between the two possibilities described soon then the rule of thumb is to use `LEX_T_SKIP` and you are safe.

The type `LEX_T_SKIP` should be used in case the token is handled due to `ProcessLexSymbol` preprocessor command and `LEX_T_SKIP` otherwise.

When the type is set `LEX_T_SKIP_SYMBOL` the lexical analyzer knows to release the string holding the symbol. If the type is `LEX_T_SKIP` only the token record is released.

If the symbol string is not released due to erroneously setting the type to `LEX_T_SKIP` instead `LEX_T_SKIP_SYMBOL` the memory will not be released until the interpreter finishes pre execution steps. So usually if you do not know how to set the type to skip a token `LEX_T_SKIP` is safe.

When processing comments the preprocessor should either use only the comments starting with the keyword `rem` or should carefully detect the comments starting with `'`.

For more information how to do it you really have to look at the function `lex_RemoveComments` in the file '`lexer.c`'.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.9 PreprocessorLexNASymbol

This entry point is used when the lexer has found a non alpha symbol. Non alpha symbols are predefined character strings, like `<>` or `<=` that contain more than one character but are not alpha characters.

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` "by value". Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.10 PreprocessorLexASymbol

This entry point is used when the lexer has found an alpha symbol. Alpha symbols are the keywords that are predefined in ScriptBasic.

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` "by value". Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.11 PreprocessorLexSymbol

This entry point is used when the lexer finds a symbol that is alphanumeric but is not predefined by ScriptBasic. These are the variables and symbols (like the statement `OPEN` opening modes.)

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` "by value". Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.12 PreprocessorLexString

This entry point is used when the preprocessor has processed a single-line string.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.13 PreprocessorLexMString

This entry point is used when the preprocessor has processed a multi-line string.

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` "by value". Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.14 PreprocessorLexInteger

This entry point is called when the lexer has processed an integer.

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` "by value". Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.15 PreprocessorLexReal

This entry point is called when the lexer has processed an real number.

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` "by value". Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.16 PreprocessorLexCharacter

This entry point is called when the lexer has processed a character.

When this entry point is called the pointer `p` points to a pointer that point to a pointer pointing to the last processed lexeme. This seems to be quite complex and uselessly complex to pass such a pointer at the firstglance. However it is not.

When the lexer builds the list of the lexemes reading the characters it creates a linked list of structures of type `Lexeme`. The lexer object field `pLexResult` points to the first element of this list. The lexer code uses a local variable named `plexLastLexeme` that first points to this pointer, and later it always points to the forward link pointer of the last element of the list.

When the preprocessor is called using this entry point this variable passed in the argument `p` `"`by value`"`. Through this pointer you can

alter the last token fields

unhook the last token and optionally hook something else on it or even

unhook any number of elements from the list that were already hooked and set the last forward link pointer as you wish

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.17 PreprocessorExStart

This entry point is used when the syntax analyzer starts.

The argument `p` points to the actual `peXobject` syntax analysis object structure.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.18 PreprocessorExStartLine

This entry point is used when the syntax analyzer starts to analyze a program line. The argument `p` points to the actual `peXobject` syntax analysis object structure. The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.19 PreprocessorExEnd

This entry point is used when the syntax analyzer has finished analyzing the basic program. The argument `p` points to the actual `peXobject` syntax analysis object structure. The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.20 PreprocessorExFinish

This entry point is called from the function `scriba_DoSyntaxAnalysis` implemented in the file '`scriba.c`' when the syntax analyzer has finished.

The only difference between this entry point and the entry point `PreprocessorExEnd` is that at this point the field `pCommandList` in the `peXobject` object structure already points to the list of nodes.

The argument `p` points to the actual `peXobject` syntax analysis object structure. The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.21 PreprocessorExStartLocal

This entry point is used when the syntax analyzer starts a new local scope. This is when a function or a sub starts.

The argument `p` points to the actual `peXobject` syntax analysis object structure. The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.22 PreprocessorExEndLocal

This entry point is used when the syntax analyzer exists a local scope. This is when a function or a sub ends.

The argument `p` points to the actual `peXobject` syntax analysis object structure. The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.23 PreprocessorExLineNode

This entry point is used when the syntax analyzer creates a new node for a basic program line.

The argument `p` points to the actual `peXobject` syntax analysis object structure. The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.24 PreprocessorExeStart

This entry point is used from the function `scriba_Run` implemented in the file '`scriba.c`' before the execution of the basic program starts.

The argument `p` points to the execution context of the basic program.

By this time most of the preprocessors should have asked the basic interpreter to unload. Only preprocessors implementing debugger, profiler or other development support functions may remain in memory and active.

At this very point debugger or other development support preprocessors may and should access the execution hook functions, like the sample debugger preprocessor does:

```
       case PreprocessorExeStart:

          pExecuteObject pEo = p;
           pDebuggerObject pDO = pEXT->pPointer;
           pEo->pHookers->hook_pointer = pEXT;
           pDO->CallStackDepth = 0;
           pDO->DbgStack = NULL;
           pDO->StackTop = NULL;
           pEo->pHookers->HOOK_ExecBefore = MyExecBefore;
           pEo->pHookers->HOOK_ExecAfter = MyExecAfter;
           pEo->pHookers->HOOK_ExecCall = MyExecCall;
           pEo->pHookers->HOOK_ExecReturn = MyExecReturn;
           *pCmd = PreprocessorContinue;
           return 0;
```

(Note that this is an example only and not the actual code. The actual code performs other tasks as well.)

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.25 PreprocessorExeFinish

This entry point is used from the function `scriba_Run` implemented in the file '`scriba.c`' after the execution of the basic program finished.

The argument `p` points to the execution context of the basic program.

By this time most of the preprocessors should have asked the basic interpreter to unload. Only preprocessors implementing debugger, profiler or other development support functions may remain in memory and active.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

5.3.26 PreprocessorExeNoRun

This entry point is used from the function `scriba_NoRun` implemented in the file '`scriba.c`' before the execution of the basic program is not started. (Note that `scriba_NoRun` performs initialization for the execution but does not start the execution.)

By this time most of the preprocessors should have asked the basic interpreter to unload. Only preprocessors implementing debugger, profiler or other development support functions may remain in memory and active.

The function has to return zero or the error code and should set the parameter `*pCmd` to `PreprocessorContinue`, `PreprocessorDone`, or `PreprocessorUnload`.

6 Compilation

ScriptBasic is written in C and thus there is a need for some C compiler to successfully install it from source. However some of the code is not maintained in C but rather in some higher level language (what an exaggeration!), which is compiled to C. To compile all the code from source you also need a functional Perl interpreter executing the version 5 of the language Perl.

Compilation is automated as much as it is possible and was highly tested on Windows NT and under Linux (Debian). Under other operating systems resembling to UNIX the compilation should be quite straight forward though may not be so seamless as it is for the ones I tested.

6.1 Compilation under UNIX

Compilation starts from the clean source package. You unzip it or gunzip untar it into the source directory. To be safe you can run

```
sh convert.sh
```

that converts all text files from Windows line feed convention to UNIX convention. If you have a tarball rather than a ZIP file then you may already have the correct line ending, but running the conversion does not hurt. Well, if you feel better by that you can run it many times. On the other hand if you uploaded the Windows source package you will not get along compiling the package without this conversion.

To compile the code you have to issue the command

```
./setup
```

This will run all the commands that are needed before compilation to generate the C source files from the real source files and also compiles the make files from their macro source. Finally it runs the compilation processes.

The program '`setup.pl`' started by the shell script '`setup`' is a huge Perl script that automates the compilation under UNIX and under Windows NT as well.

To install the compiled code the script '`setup.pl`' has to be started again, this time using the argument:

```
./setup -install
```

Although its use may imply it does not install ScriptBasic. It only creates the file '`install.sh`' that you can run later any time

```
$ su
Password: **********
# ./install.sh
```

from the root account. It is recommended that you examine and understand the content of the script before executing it. It actually installs ScriptBasic, compiles the configuration information, stops the Eszter SB Application Engine and many other things.

I said >>only creates the file ...<< in double quotes, because it does many things before actually creating the file. It interactively asks you about the destination directories, installs an experimental installation of ScriptBasic locally to run some test programs, examines which modules were compiled fine and so on. It even tests the possible maximal value for the configuration key `maxlevel` that limits the maximal recursive function call level inside ScriptBasic. To do so it creates a configuration that does not limit the depth and runs a test program until it crashes. The test program in each level of depth opens a file and appends a single byte to the file. Finally the length of the file gives the maximal possible recursive function depth on your installation.