ScriptBasic Source Files

Peter Verhas

Short Contents

Table of Contents

1 Introduction

This file contains the source code documentation. Most of this information is also compiled into the developers guide or into other documentation.

1.1 scriba.c

1.1.1 scriba_new()

To create a new `SbProgram` object you have to call this function. The two arguments should point to `malloc` and `free` or similar functions. All later memory allocation and releasing will be performed using these functions.

Note that this is the only function that does not require a pointer to an `SbProgram` object.

```
SCRIBA_MAIN_LIBSPEC pSbProgram scriba_new(void * (*maf)(size_t),
                        void   (*mrf)(void *)
);
```

1.1.2 scriba_destroy()

After a ScriptBasic program was successfully execued and there is no need to run it anymore call this function to release all memory associated with the code.

```
void scriba_destroy(pSbProgram pProgram
);
```

1.1.3 scriba_NewSbData()

Allocate and return a pointer to the allocated `SbData` structure.

This structure can be used to store ScriptBasic variable data, long, double or string. This function is called by other functions from this module. Usually the programmer, who embeds ScriptBasic will rarely call this function directly. Rather he/she will use See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩ (as an example) that creates a variable capable holding a `long`, sets the type to be `SBT_LNG` and stores initial value.

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

```
pSbData scriba_NewSbData(pSbProgram pProgram
);
```

1.1.4 scriba_InitSbData()

This function initializes an SbData structure to hold undef value. This function should be used to initialize an allocated `SbData` memory structure. This function internally is called by See ⟨undefined⟩ [scriba_NewSbData()], page ⟨undefined⟩.

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

```
void scriba_InitSbData(pSbProgram pProgram,
                       pSbData p
);
```

1.1.5 scriba_UndefSbData()

This function sets an `SbData` structure to hold the undefined value.

This function should should not be used instead of See ⟨undefined⟩ [scriba_InitSbData()], page ⟨undefined⟩. While that function should be used to inititalize the memory structure this function should be used to set the value of an alreasdy initialized and probably used `SbData` variable to `undef`.

The difference inside is that if the `SbData` structure is a string then this function releases the memory occupied by the string, while See ⟨undefined⟩ [scriba_InitSbData()], page ⟨undefined⟩ does not.

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

```
void scriba_UndefSbData(pSbProgram pProgram,
                        pSbData p
);
```

1.1.6 scriba_NewSbLong()

This function allocates and returns a pointer pointing to a structure of type `SbData` holding a `long` value. If the allocation failed the return value is `NULL`. If the memory allocation was successful the allocated structure will have the type `SBT_LONG` and will hold the initial value specified by the argument `lInitValue`.

```
pSbData scriba_NewSbLong(pSbProgram pProgram,
                         long lInitValue
);
```

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()],

page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

1.1.7 scriba_NewSbDouble()

This function allocates and returns a pointer pointing to a structure of type `SbData` holding a `double` value. If the allocation failed the return value is `NULL`. If the memory allocation was successful the allocated structure will have the type `SBT_DOUBLE` and will hold the initial value specified by the argument `dInitValue`.

```
pSbData scriba_NewSbDouble(pSbProgram pProgram,
                           double dInitValue
);
```

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

1.1.8 scriba_NewSbUndef()

This function allocates and returns a pointer pointing to a structure of type `SbData` holding an `undef` value. If the allocation failed the return value is `NULL`. If the memory allocation was successful the allocated structure will have the type `SBT_UNDEF`.

```
pSbData scriba_NewSbUndef(pSbProgram pProgram
);
```

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

1.1.9 scriba_NewSbString()

This function allocates and returns a pointer pointing to a structure of type `SbData` holding a string value. If the allocation failed the return value is `NULL`. If the memory allocation was successful the allocated structure will have the type `SBT_STRING` and will hold the initial value specified by the argument `pszInitValue`.

```
pSbData scriba_NewSbString(pSbProgram pProgram,
                           char *pszInitValue
);
```

Note on ZCHAR termination:

The init value `pszInitValue` should be a zchar terminated string. Note that ScriptBasic internally stores the strings as series byte and the length of the string without any

terminating zchar. Therefore the length of the string that is stored should have been `strlen(pszInitValue)`. This does not contain the terminating zchar.

In reality however we allocate an extra byte that stores the zchar, but the size of the string is one character less. Therefore ScriptBasic routines will recognize the size of the string correct and also the caller can use the string using the macro `scriba_GetString` as a zchar terminated C string. This requires an extra byte of storage for each string passed from the embedding C application to ScriptBasic, but saves a lot of hedeache and also memory copy when the string has to be used as a zchar terminated string.

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

1.1.10 scriba_NewSbBytes()

This function allocates and returns a pointer pointing to a structure of type `SbData` holding a string value. If the allocation failed the return value is `NULL`. If the memory allocation was successful the allocated structure will have the type `SBT_STRING` and will hold the initial value specified by the argument `pszInitValue` of the length `len`.

```
pSbData scriba_NewSbBytes(pSbProgram pProgram,
                          unsigned long len,
                          unsigned char *pszInitValue
);
```

This function allocates `len+1` number of bytes data and stores the initial value pointed by `pszInitValue` in it.

The extra plus one byte is an extra terminating zero char that may help the C programmers to handle the string in case it is not binary. Please also read the not on the terminating ZChar in the function See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩.

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

1.1.11 scriba_DestroySbData()

Call this function to release the memory that was allocated by any of the `NewSbXXX` functions. This function releases the memory and also cares to release the memory occupied by the characters in case the value had the type `SBT_STRING`.

```
void scriba_DestroySbData(pSbProgram pProgram,
                          pSbData p
);
```

See also See ⟨undefined⟩ [scriba_NewSbLong()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbDouble()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbUndef()],

page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbString()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NewSbBytes()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DestroySbData()], page ⟨undefined⟩.

1.1.12 scriba_PurgeReaderMemory()

Call this function to release all memory that was allocated by the reader module. The memory data is needed so long as long the lexical analyzer has finished.

```
void scriba_PurgeReaderMemory(pSbProgram pProgram
);
```

1.1.13 scriba_PurgeLexerMemory()

```
void scriba_PurgeLexerMemory(pSbProgram pProgram
);
```

1.1.14 scriba_PurgeSyntaxerMemory()

```
void scriba_PurgeSyntaxerMemory(pSbProgram pProgram
);
```

1.1.15 scriba_PurgeBuilderMemory()

```
void scriba_PurgeBuilderMemory(pSbProgram pProgram
);
```

1.1.16 scriba_PurgePreprocessorMemory()

This function purges the memory that was needed to run the preprocessors.

```
void scriba_PurgePreprocessorMemory(pSbProgram pProgram
);
```

1.1.17 scriba_PurgeExecuteMemory()

This function purges the memory that was needed to execute the program, but before that it executes the finalization part of the execution.

```
void scriba_PurgeExecuteMemory(pSbProgram pProgram
);
```

1.1.18 scriba_SetFileName()

Call this function to set the file name where the source informaton is. This file name is used by the functions See ⟨undefined⟩ [scriba_LoadBinaryProgram()], page ⟨undefined⟩ and See ⟨undefined⟩ [scriba_LoadSourceProgram], page ⟨undefined⟩ as well as error reporting functions to display the location of the error.

```
int scriba_SetFileName(pSbProgram pProgram,
                       char *pszFileName
);
```

The argument `pszFileName` should be zchar terminated string holding the file name.

1.1.19 scriba_GettingConfiguration()

See ⟨undefined⟩ [scriba_LoadConfiguration()], page ⟨undefined⟩ and See ⟨undefined⟩ [scriba_InheritConfiguration()], page ⟨undefined⟩ can be used to specify configuration information for a ScriptBasic program. Here we describe the differences and how to use the two functions for single-process single-basic and for single-process multiple-basic applications.

To execute a ScriptBasic program you usually need configuration information. The configuration information for the interpreter is stored in a file. The function See ⟨undefined⟩ [scriba_LoadConfiguration()], page ⟨undefined⟩ reads the file and loads it into memory into the `SbProgram` object. When the object is destroyed the configuration information is automatically purged from memory.

Some implementations like the Eszter SB Engine variation of ScriptBasic starts several interpreter thread within the same process. In this case the configuration information is read only once and all the running interpreters share the same configuration information.

To do this the embedding program has to create a pseudo `SbProgram` object that does not run any ScriptBasic program, but is used only to load the configuration information calling the function See ⟨undefined⟩ [scriba_LoadConfiguration()], page ⟨undefined⟩. Other `SbProgram` objects that do intepret ScriptBasic program should inherit this configuration calling the function See ⟨undefined⟩ [scriba_InheritConfiguration()], page ⟨undefined⟩. When a `SbProgram` object is destroyed the configuration is not destroyed if that was inherited belonging to a different object. It remains in memory and can later be used by other intrepreter instances.

Inheriting the configuration is fast because it does not require loading the configuration information from file. This is essentially sets a pointer in the internal interpreter structure to point to the configuration information held by the other object and all the parallel running interpreters structures point to the same piece of memory holding the common configuration information.

See the configuration handling functions See ⟨undefined⟩ [scriba_LoadConfiguration()], page ⟨undefined⟩ and See ⟨undefined⟩ [scriba_InheritConfiguration()], page ⟨undefined⟩.

1.1.20 scriba_LoadConfiguration()

This function should be used to load the configuration information from a file.

The return value is zero on success and the error code when error happens.

```
int scriba_LoadConfiguration(pSbProgram pProgram,
                             char *pszForcedConfigurationFileName
);
```

1.1.21 scriba_GetConfigFileName()

This function tells whet the configuration file is. There is no need to call this function to read the configuration file. This is needed only when the main program want to manipulate the configuration file in some way. For example the command line version of ScriptBasic uses this function when the option `-k` is used to compile a configuration file.

The first argument has to be a valid ScriptBasic program object. The second argument should point to a valid `char *` pointer that will get the pointer value to the configuration file name after the function returns.

```
int scriba_GetConfigFileName(pSbProgram pProgram,
                             char **ppszFileName
);
```

The function returns zero if no error happens, or the error code.

1.1.22 scriba_InheritConfiguration()

Use this function to get the configuration from another program object.

The return value is zero on success and error code if error has happened.

```
int scriba_InheritConfiguration(pSbProgram pProgram,
                                pSbProgram pFrom
);
```

1.1.23 scriba_InitModuleInterface()

Initialize the Support Function Table of a process level ScriptBasic program object to be inherited by other program objects. If you read it first time, read on until you understand what this function really does and rather how to use it!

This is going to be a bit long, but you better read it along with the documentation of the function See ⟨undefined⟩ [scriba_InheritModuleInterface()], page ⟨undefined⟩.

This function is needed only for programs that are

multi thread running several interpreters simultaneous in a single process

support modules like the sample module `mt` that support multithread behaviour and need to implement worker thread needing call-back functions.

You most probably know that modules can access system and ScriptBasic fucntions via a call-back table. That is a huge `struct` containing pointers to the functions that ScriptBasic implements. This is the `ST` (aka support table).

This helps module writers to write system independent code as well as to access ScriptBasic functions easily. On the other hand modules are also free to alter this table and because many functions, tough not all are called via this table by ScriptBasic itself a module may alter the core behavior of ScriptBasic.

For this reason each interpreter has its own copy of `ST`. This means that if an interpreter alters the table it has no effect on another interpreter running in the same process in anther thread.

This is fine so far. How about modules that run asynchronous threads? For example the very first interpter thread that uses the module `mt` starts in the initialization a thread that later deletes all sessions that time out. This thread lives a long life.

The thread that starts the worker thread is an interpreter thread and has its own copy of the `ST`. The thread started asynchronous however should not use this `ST` because the table is purged from memory when the interpreter instance it blelonged to finishes.

To have `ST` for worker threads there is a need for a program object that is not purged from memory so long as long the process is alive. Fortunately there is such an object: the configuration program object. Configuration is usually read only once by multi-thread implementations and the same configuration information is shared by the serveral threads. The same way the several program objects may share a `ST`.

The difference is that configuration is NOT altered by the interpreter or by any module in any way but `ST` may. Thus each execution object has two pointers: `pST` and `pSTI`. While `pST` points to the support table that belongs to the interpreter instance the secondpointer `pSTI` points to a `ST` that is global for the whole process and is permanent. This `ST` is to be used by worker threads and should not be altered by the module without really good reason.

Thus: Don't call this function for normal program objects! For usualy program objects module interface is automatically initialized when the first module function is called. Call this function to initialize a `ST` for a pseudo program object that is never executed but rather used to inherit this `ST` for worker threads.

```
int scriba_InitModuleInterface(pSbProgram pProgram
);
```

1.1.24 scriba_InheritModuleInterface()

Inherit the support function table (`ST`) from another program object.

Note that the program object is going to initialize its own `ST` the normal way. The inherited `ST` will only be used by worker threads that live a long life and may exist when the initiating interpreter thread already exists.

For further information please read the description of the function See ⟨undefined⟩ [scriba_InitModuleInterface()], page ⟨undefined⟩.

```
int scriba_InheritModuleInterface(pSbProgram pProgram,
                                  pSbProgram pFrom
);
```

1.1.25 scriba_InheritExecuteObject()

```
int scriba_InheritExecuteObject(pSbProgram pProgram,
                                pSbProgram pFrom
);
```

1.1.26 scriba_SetProcessSbObject()

Use this program in multi-thread environment to tell the actual interpreter which object is the process level pseudo object that

holds the shared (among interpreter thread objects) configuration information (see See ⟨undefined⟩ [scriba_InheritConfiguration()], page ⟨undefined⟩)

holds the process level module interface (see See ⟨undefined⟩ [scriba_InheritModuleInterface()], page ⟨undefined⟩)

holds the list of loaded modules that are not unloaded by the thread loaded the module

If the embeddingprogram calls this function there is no need to call See ⟨undefined⟩ [scriba_InheritConfiguration()], page ⟨undefined⟩ and See ⟨undefined⟩ [scriba_InheritModuleInterface()], page ⟨undefined⟩. This function call does all those tasks and also other things.

```
int scriba_SetProcessSbObject(pSbProgram pProgram,
                              pSbProgram pProcessObject
);
```

1.1.27 scriba_ShutdownMtModules()

A multi threaded application should call this function for the process SB object when the process finishes. Calling this function will call each of the shutdown functions of those external modules that decided to keep in memory and export the shutdown function named `shutmodu`. This allows these modules to gracefully shut down their operation. As an example cached data can be written to disk, or database connections can be closed.

```
int scriba_ShutdownMtModules(pSbProgram pProgram
);
```

1.1.28 scriba_SetCgiFlag()

You can call this function to tell the reporting subsystem that this code runs in a CGI environment and therefore it should format error messages according to the CGI standard sending to the standard output including HTTP headers and HTML code pieces.

```
void scriba_SetCgiFlag(pSbProgram pProgram
);
```

1.1.29 scriba_SetReportFunction()

This function should be used to set the report function for a program. The report function is used to send info, warning, error, fatal and internal error messages to the user.

In case you want to implement a specific report function see the sample implementation in the file `report.c`. The documentation of the function `report_report` describes not only the details of the sample implementation but also the implementation requests for other reporting functions.

```
void scriba_SetReportFunction(pSbProgram pProgram,
                              void *fpReportFunction
);
```

1.1.30 scriba_SetReportPointer()

This pointer will be passed to the reporting function. The default reporting uses this pointer as a `FILE *` pointer. The default value for this pointer is `stderr`.

Other implementations of the reporting function may use this pointer according their needs. For example the WIN32 IIS ISAPI implementation uses this pointer to point to the extension controll block structure.

```
void scriba_SetReportPointer(pSbProgram pProgram,
                             void *pReportPointer
);
```

1.1.31 scriba_SetStdin()

You can call this function to define a special standard input function. This pointer should point to a function that accepts a `void *` pointer as argument. Whenever the ScriptBasic program tries to read from the standard input it calls this function pasing the embedder pointer as argument.

If the `stdin` function is not defined or the parameter is `NULL` the interpreter will read the normal `stdin` stream.

```
void scriba_SetStdin(pSbProgram pProgram,
                     void *fpStdinFunction
);
```

1.1.32 scriba_SetStdout()

You can call this function to define a special standard output function. This pointer should point to a function that accepts a `(char, void *)` arguments. Whenever the ScriptBasic program tries to send a character to the standard output it calls this function. The first parameter is the character to write, the second is the embedder pointer.

If the standard output function is not defined or the parameter is `NULL` the interpreter will write the normal `stdout` stream.

```
void scriba_SetStdout(pSbProgram pProgram,
                      void *fpStdoutFunction
);
```

1.1.33 scriba_SetEmbedPointer()

This function should be used to set the embed pointer.

The embed pointer is a pointer that is not used by ScriptBasic itself. This pointer is remembered by ScriptBasic and is passed to call-back functions. Like the standard input, output and environment functions that the embedding application may provide this pointer is also available to external modules implemented in C or other compiled language in DLL or SO files.

The embedder pointer should usually point to the `struct` of the thread local data. For example the Windows NT IIS variation of ScriptBasic sets this variable to point to the extension control block.

If this pointer is not set ScriptBasic will pass `NULL` pointer to the extensions and to the call-back function.

```
void scriba_SetEmbedPointer(pSbProgram pProgram,
                            void *pEmbedder
);
```

1.1.34 scriba_SetEnvironment()

You can call this function to define a special environment query function. This pointer should point to a function that accepts a `(void *, char *, long )` arguments.

Whenever the ScriptBasic program tries to get the value of an enviroment variable it calls this function. The first argument is the embedder pointer.

The second argument is the name of the environment variable to retrieve or `NULL`.

The third argument is either zero or is the serial number of the environment variable.

ScriptBasic never calls this function with both specifying the environment variable name and the serial number.

The return value of the function should either be `NULL` or should point to a string that holds the zero character terminated value of the environment variable. This string is not changed by ScriptBasic.

If the special environment function is not defined or is `NULL` ScriptBasic uses the usual environment of the process calling the system function`getenv`.

```
void scriba_SetEnvironment(pSbProgram pProgram,
                           void *fpEnvirFunction
);
```

For a good example of a self-written environment function see the source of the Eszter SB Engine that alters the environment function so that the ScriptBasic programs feel as if they were executed in a real CGI environment.

1.1.35 scriba_LoadBinaryProgramWithOffset()

Use this function to load ScriptBasic program from a file that is already compiled into internal form, and the content of the program is starting on `lOffset`

The return value is the number of errors (hopefully zero) during program load.

```
int scriba_LoadBinaryProgramWithOffset(pSbProgram pProgram,
                                       long lOffset,
                                       long lEOFfset
);
```

Before calling this function the function See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩ should have been called specifying the file name.

1.1.36 scriba_LoadBinaryProgram()

Use this function to load ScriptBasic program from a file that is already compiled into internal form.

The return value is the number of errors (hopefully zero) during program load.

```
int scriba_LoadBinaryProgram(pSbProgram pProgram
);
```

Before calling this function the function See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩ should have been called specifying the file name.

1.1.37 scriba_InheritBinaryProgram()

Use this function in application that keeps the program code in memory.

```
int scriba_InheritBinaryProgram(pSbProgram pProgram,
                                pSbProgram pFrom
);
```

The function inherits the binary code from the program object `pFrom`. In server type applications the compiled binary code of a BASIC program may be kept in memory. To do this a pseudo program object should be created that loads the binary code and is not destroyed.

The program object used to execute the code should inherit the binary code from this pseudo object calling this function. This is similar to the configuration inheritance.

1.1.38 scriba_LoadInternalPreprocessor()

This function can be used by embedding applications to load an internal preprocessor into the interpereter. Note that preprocessors are usually loaded by the reader module when a `preprocess` statement is found. However some preprocessors in some variation of the interpreter may be loaded due to configuration or command line option and not because the source requests it.

The preprocessors that are requested to be loaded because the source contains a `preprocess` line usually implement special language fetures. The preprocessors that are loaded independent of the source because command line option or some other information tells the variation to call this function are usually debuggers, profilers.

(To be honest, by the time I write it there is no any internal preprocessors developed except the test one, but the statement above will become true.)

```
int scriba_LoadInternalPreprocessor(pSbProgram pProgram,
                            char *ppszPreprocessorName[]
);
```

The first argument is the program object. If the program object does not have a preprocessor object the time it is called the preprocessor object is created and initiated.

The second argument is the array of names of the preprocessor as it is present in the configuration file. This is not the name of the DLL/SO file, but rather the symbolic name, which is associated with the file. The final element of the array has to be `NULL`.

The return value is zero or the error code.

1.1.39 scriba_ReadSource()

Loads the source code of a ScriptBasic program from a text file.

The return code is the number of errors happened during read.

```
int scriba_ReadSource(pSbProgram pProgram
);
```

Do not get confused! This function only reads the source. Does not compile it. You will usually need See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨undefined⟩ that does reading, analyzing, building and all memory releases leaving finally a ready-to-run code in memory.

Before calling this function the function See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩ should have been called specifying the file name.

See also See ⟨undefined⟩ [scriba_ReadSource()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoLexicalAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoSyntaxAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_BuildCode()], page ⟨undefined⟩.

1.1.40 scriba_DoLexicalAnalysis()

This function performs lexical analysis after the source file has beed read.

This function is rarely needeed by applicationdevelopers. See See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨undefined⟩ instead.

```
int scriba_DoLexicalAnalysis(pSbProgram pProgram
);
```

See also See ⟨undefined⟩ [scriba_ReadSource()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoLexicalAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoSyntaxAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_BuildCode()], page ⟨undefined⟩.

1.1.41 scriba_DoSyntaxAnalysis()

This function performs syntax analysis after the lexical analysis has been finished.

This function is rarely needeed by applicationdevelopers. See See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨undefined⟩ instead.

```
int scriba_DoSyntaxAnalysis(pSbProgram pProgram
);
```

See also See ⟨undefined⟩ [scriba_ReadSource()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoLexicalAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoSyntaxAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_BuildCode()], page ⟨undefined⟩.

1.1.42 scriba_BuildCode()

This function builds the finall ready-to-run code after the syntax analisys has been finished.

This function is rarely needeed by applicationdevelopers. See See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨undefined⟩ instead.

```
int scriba_BuildCode(pSbProgram pProgram
);
```

See also See ⟨undefined⟩ [scriba_ReadSource()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoLexicalAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoSyntaxAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_BuildCode()], page ⟨undefined⟩.

1.1.43 scriba_IsFileBinaryFormat()

This function decides if a file is a correct binary format ScriptBasic code file and returns true if it is binary. If the file is a ScriptBasic source file or an older version binary of ScriptBasic or any other file it returns zero.

This function just calls the function `build_IsFileBinaryFormat`

```
int scriba_IsFileBinaryFormat(pSbProgram pProgram
);
```

1.1.44 scriba_GetCacheFileName()

Calculate the name of the cache file for the given source file name and store the calculated file name in the program object.

```
int scriba_GetCacheFileName(pSbProgram pProgram
);
```

The program returns zero or the error code. It returns `SCRIBA_ERROR_FAIL` if there is no cache directory configured.

The code uses a local buffer of length 256 bytes. The full cached file name should fit into this otherwise the program will return `SCRIBA_ERROR_BUFFER_SHORT`.

The code does not check if there exists an appropriate cache directory or file. It just calculates the file name.

1.1.45 scriba_UseCacheFile()

Call this function to test that the cache file is usable. This function calls the function See ⟨undefined⟩ [scriba_GetCacheFileName()], page ⟨undefined⟩ to calculate the cache file name.

If

the cache file exists

is newer than the source file set by See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩

is a correct ScriptBasic binary file

then this function alters the source file name property (`pszFileName`) of the program object so that the call to See ⟨undefined⟩ [scriba_LoadBinaryProgram()], page ⟨undefined⟩ will try to load the cache file.

```
int scriba_UseCacheFile(pSbProgram pProgram
);
```

The function returns zero or the error code. The function returns `SCRIBA_ERROR_FAIL` in case the cache file is old, or not valid. Therefore returning a positive value does not neccessarily mean a hard error.

1.1.46 scriba_SaveCacheFile()

Call this function to generate a cache file after a successful program compilation.

```
int scriba_SaveCacheFile(pSbProgram pProgram
);
```

The function returns zero (`SCRIBA_ERROR_SUCCESS`) if there was no error. This does not mean that the cache file was saved. If there is no cache directory configured doing nothing is success.

Returning any positive error code means that ScriptBasic tried to write a cache file but it could not.

1.1.47 scriba_RunExternalPreprocessor()

This function should be called to execute external preprocessors.

This function does almost nothing else but calls the function `epreproc()`.

```
int scriba_RunExternalPreprocessor(pSbProgram pProgram,
                                   char **ppszArgPreprocessor
);
```

The argument `ppszArgPreprocessor` should point to a string array. This string array should contain the configured names of the preprocessors that are applied one after the other in the order they are listed in the array.

Note that this array should contain the symbolic names of the preprocessors. The actual preprocessor executable programs, or command lines are defined in the configuration file.

After calling this function the source file name property of the program object (`pszFileName`) is also modified so that it points to the result of the preprocessor. This means that after the successful return of this function the application may immediately call See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨undefined⟩.

If there is any error during the preprocessor execution the function returns some error code (returned by `epreproc`) otherwise the return value is zero.

1.1.48 scriba_SaveCode()

Call this function to save the compiled byte code of the program into a specific file. This function is called by the function See ⟨undefined⟩ [scriba_SaveCacheFile()], page ⟨undefined⟩.

```
int scriba_SaveCode(pSbProgram pProgram,
                    char *pszCodeFileName
);
```

The function does nothing else, but calls `build_SaveCode`.

The return code is zero or the error code returned by `build_SaveCode`.

1.1.49 scriba_SaveCCode()

```
void scriba_SaveCCode(pSbProgram pProgram,
                      char *pszCodeFileName
);
```

1.1.50 scriba_SaveECode()

```
void scriba_SaveECode(pSbProgram pProgram,
                      char *pszInterpreter,
                      char *pszCodeFileName
);
```

1.1.51 scriba_LoadSourceProgram()

Call this function to load a BASIC program from its source format after optionally checking that there is no available cache file and after executing all required preprocessors. This function calls See ⟨undefined⟩ [scriba_ReadSource()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoLexicalAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_DoSyntaxAnalysis()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_BuildCode()], page ⟨undefined⟩, and also releases the memory that was needed only for code building calling See ⟨undefined⟩ [scriba_PurgeReaderMemory()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_PurgeLexerMemory()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_PurgeSyntaxerMemory()], page ⟨undefined⟩.

After the successful completion of this program the BASIC program is in the memory in the ready-to-run state.

```
int scriba_LoadSourceProgram(pSbProgram pProgram
);
```

Before calling this function the function See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩ should have been called specifying the file name.

The return value is zero (`SCRIBA_ERROR_SUCCESS`) or the error code returned by the underlying layer that has detected the error.

1.1.52 scriba_LoadProgramString()

Use this function to convert a string containing a BASIC program that is already in memory to ready-to-run binary format. This function is same as See ⟨undefined⟩ [scriba_LoadSourceProgram()], page ⟨undefined⟩ except that this function reads the source code from a string instead of a file.

```
int scriba_LoadProgramString(pSbProgram pProgram,
                             char *pszSourceCode,
                             unsigned long cbSourceCode
);
```

The argument `pProgram` is the program object. The argument `pszSourceCode` is the BASIC program itself in text format. Because the source code may contain ZCHAR just for any chance the caller has to provide the number of characters in the buffer via the argument `cbSourceCode`. In case the source program is zero terminated the caller can simply say `strlen(pszSourceCode)` to give this argument.

Before calling this function the function See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩ may be called. Altough the source code is read from memory and thus there is no source file the BASIC program may use the command `include` or `import` that includes another source file after reading the code. If the program does so the reader functions need to know the actual file name of the source code to find the file to be included. To help this process the caller using this function may set the file name calling See ⟨undefined⟩ [scriba_SetFileName()], page ⟨undefined⟩. However that file is never used and need not even exist. It is used only to calculate the path of included files that are specified using relative path.

The return value is zero (`SCRIBA_ERROR_SUCCESS`) or the error code returned by the underlying layer that has detected the error.

1.1.53 scriba_Run()

Call this function to execute a program. Note that you can call this function many times. Repetitive execution of the same program will execute the ScriptBasic code again and again with the global variables keeping their values.

If you want to reset the global variables you have to call See ⟨undefined⟩ [scriba_ResetVariables()], page ⟨undefined⟩.

There is no way to keep the value of the local variables.

The argument `pszCommandLineArgument` is the command part that is passed to the BASIC program.

```
int scriba_Run(pSbProgram pProgram,
               char *pszCommandLineArgument
);
```

The return value is zero in case of success or the error code returned by the underlying execution layers.

Note that you can not call BASIC subroutines or functions without initializations that `scriba_Run` performs. You also can not access global variables. Therefore you either have

to call `scriba_Run` or its brother See ⟨undefined⟩ [scriba_NoRun()], page ⟨undefined⟩ that performs the initializations without execution.

You also have to call See ⟨undefined⟩ [scriba_NoRun()], page ⟨undefined⟩ if you want to execute a program with some global variables having preset values that you want to set from the embedding C program. In that case you have to call See ⟨undefined⟩ [scriba_NoRun()], page ⟨undefined⟩ then one or more times See ⟨undefined⟩ [scriba_SetVariable()], page ⟨undefined⟩ and finally `Run`.

1.1.54 scriba_NoRun()

In case the embedding program want to set global variables and execute subroutines without or before starting the main program it has to call this function first. It does all the initializations that are done by See ⟨undefined⟩ [scriba_Run()], page ⟨undefined⟩ except that it does not actually execute the program.

After calling this function the main program may access global variables and call BASIC functions.

```
int scriba_NoRun(pSbProgram pProgram
);
```

See also See ⟨undefined⟩ [scriba_Run()], page ⟨undefined⟩.

1.1.55 scriba_ResetVariables()

Call this function if you want to execute a program object that was already executed but you do not want the global variables to keep their value they had when the last execution of the BASIC code finished.

Global variables in ScriptBasic are guaranteed to be `undef` before they get any other value and some programs depend on this.

```
void scriba_ResetVariables(pSbProgram pProgram
);
```

See also See ⟨undefined⟩ [scriba_SetVariable()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_Run()], page ⟨undefined⟩, See ⟨undefined⟩ [scriba_NoRun()], page ⟨undefined⟩.

1.1.56 scriba_Call()

This function can be used to call a function or subroutine. This function does not get any arguments and does not provide any return value.

```
int scriba_Call(pSbProgram pProgram,
                unsigned long lEntryNode
);
```

The return value is zero or the error code returned by the interpreter.

Note on how to get the Entry Node value:

The argument `lEntryNode` should be the node index of the subroutine or function that we want to execute. This can be retrieved using the function See ⟨undefined⟩ [scriba_LookupFunctionByName()], page ⟨undefined⟩ if the name of the function or

subroutine is know. Another method is that the BASIC program stored this value in some global variables. BASIC programs can access this information calling the BASIC function `Address( f() )`.

1.1.57 scriba_CallArg()

This function can be used to call a function or subroutine with arguments passed by value. Neither the return value of the SUB nor the modified argument variables are not accessible via this function. `CallArg` is a simple interface to call a ScriptBasic subroutine or function with argument.

```
int scriba_CallArg(pSbProgram pProgram,
                   unsigned long lEntryNode,
                   char *pszFormat, ...
);
```

Arguments

`pProgram` is the class variable.

`lEntryNode` is the start node of the SUB. (See See ⟨undefined⟩ [scriba_Call()], page ⟨undefined⟩ note on how to get the entry node value.)

`pszFormat` is a format string that defines the rest of the areguments

The format string is case insensitive. The characters `u`, `i`, `r`, `b` and `s` have meaning. All other characters are ignored. The format characters define the type of the arguments from left to right.

`u` means to pass an `undef` to the SUB. This format character is exceptional that it does not consume any function argument.

`i` means that the next argument has to be `long` and it is passed to the BASIC SUB as an integer.

`r` means that the next argument has to be `double` and it is passed to the BASIC SUB as a real.

`s` means that the next argument has to be `char *` and it is passed to the BASIC SUB as a string.

`b` means that the next two arguments has to be `long cbBuffer` and `unsigned char *Buffer`. The `cbBuffer` defines the length of the `Buffer`.

Note that this SUB calling function is a simple interface and has no access to the modified values of the argument after the call or the return value.

If you need any of the functionalities that are not implemented in this function call a more sophisticated function.

Example:

```
iErrorCode = scriba_CallArg(&MyProgram,lEntry,"i i s d",13,22,"My string.",54.12)
```

1.1.58 scriba_DestroySbArgs()

This function can be used to release the memory used by arguments created by the function See ⟨undefined⟩ [scriba_NewSbArgs()], page ⟨undefined⟩.

```
void scriba_DestroySbArgs(pSbProgram pProgram,
                          pSbData Args,
                          unsigned long cArgs
);
```

Arguments:

`pProgram` class variable

`Args` pointer returned by See ⟨undefined⟩ [scriba_NewSbArgs()], page ⟨undefined⟩

`cArgs` the number of arguments pointed by `Args`

1.1.59 scriba_NewSbArgs()

Whenever you want to handle the variable values that are returned by the scriba subroutine you have to call See ⟨undefined⟩ [scriba_CallArgEx()], page ⟨undefined⟩. This function needs the arguments passed in an array of `SbDtata` type.

This function is a usefuly tool to convert C variables to an array of `SbData`

```
pSbData scriba_NewSbArgs(pSbProgram pProgram,
                         char *pszFormat, ...
);
```

The arguments passed are

`pProgram` is the class variable

`pszFormat` is the format string

The format string is case insensitive. The characters `u`, `i`, `r`, `b` and `s` have meaning. All other characters are ignored. The format characters define the type of the arguments from left to right.

`u` means to pass an `undef` to the SUB. This format character is exceptional that it does not consume any function argument.

`i` means that the next argument has to be `long` and it is passed to the BASIC SUB as an integer.

`r` means that the next argument has to be `double` and it is passed to the BASIC SUB as a real.

`s` means that the next argument has to be `char *` and it is passed to the BASIC SUB as a string.

`b` means that the next two arguments has to be `long cbBuffer` and `unsigned char *Buffer`. The `cbBuffer` defines the leng of the `Buffer`.

Example:

```
pSbData MyArgs;
```

```
MyArgs = scriba_NewSbArgs(pProgram,"i i r s b",13,14,3.14,"string",2,"two charact
if( MyArgs == NULL )error("memory alloc");

scriba_CallArgEx(pProgram,lEntry,NULL,5,MyArgs);
```

This example passes five arguments to the ScriptBasic subroutine. Note that the last one is only two character string, the rest of the characters are ignored.

1.1.60 scriba_CallArgEx()

This is the most sophisticated function of the ones that call a ScriptBasic subroutine. This function is capable handling parameters to scriba subroutines, and returning the modified argument variables and the return value.

```
int scriba_CallArgEx(pSbProgram pProgram,
                     unsigned long lEntryNode,
                     pSbData ReturnValue,
                     unsigned long cArgs,
                     pSbData Args
);
```

The arguments:

`pProgram` is the program object pointer.

`lEntryNode` is the entry node index where the BASIC subroutine or function starts (See See ⟨undefined⟩ [scriba_Call()], page ⟨undefined⟩ note on how to get the entry node value.)

`ReturnValue` is the return value of the function or subroutine

`cArgs` is the number of argments passed to the function

`Args` argument data array

1.1.61 scriba_LookupFunctionByName()

This function should be used to get the entry point of a function knowing the name of the function. The entry point should not be treated as a numerical value rather as a handle and to pass it to functions like See ⟨undefined⟩ [scriba_CallArgEx()], page ⟨undefined⟩.

```
long scriba_LookupFunctionByName(pSbProgram pProgram,
                                 char *pszFunctionName
);
```

The return value of the function is the entry node index of the function named or zero if the function is not present in the program.

1.1.62 scriba_LookupVariableByName()

This function can be used to get the serial number of a global variable knowing the name of the variable.

Note that all variables belong to a name space. Therefore if you want to retrieve the global variable `foo` you have to name it `main::foo`.

```
long scriba_LookupVariableByName(pSbProgram pProgram,
                                 char *pszVariableName
);
```

The return value is the serial number of the global avriable or zero if there is no variable with that name.

Note that the second argument, the name of the global variable, is not going under the usual name space manipulation. You have to specify the variable name together with the name space. For example the variable `a` will not be found, but the variable `main::a` will be.

1.1.63 scriba_GetVariableType()

Get the type of the value that a variable is currently holding. This value can be

`SBT_UNDEF`

`SBT_DOUBLE`

`SBT_LONG`

`SBT_STRING`

```
long scriba_GetVariableType(pSbProgram pProgram,
                            long lSerial
);
```

The argument `lSerial` should be the serial number of the variable as returned by See ⟨undefined⟩ [scriba_LookupVariableByName()], page ⟨undefined⟩.

If there is no variable for the specified serian number (`lSerial` is not positive or larger than the number of variables) the function returns `SBT_UNDEF`.

1.1.64 scriba_GetVariable()

This function retrieves the value of a variable. A new `SbData` object is created and the pointer to it is returned in `pVariable`. This memory space is automatically reclaimed when the program object is destroyed or the function `DestroySbData` can be called.

```
int scriba_GetVariable(pSbProgram pProgram,
                       long lSerial,
                       pSbData *pVariable
);
```

The argument `lSerial` should be the serial number of the global variable as returned by See ⟨undefined⟩ [scriba_LookupVariableByName()], page ⟨undefined⟩.

The funtion returns `SCRIBA_ERROR_SUCCESS` on success,

`SCRIBA_ERROR_MEMORY_LOW` if the data cannot be created or

`SCRIBA_ERROR_FAIL` if the parameter `lSerial` is invalid.

1.1.65 scriba_SetVariable()

This function sets the value of a global BASIC variable. You can call this function after executing the program before it is reexecuted or after successfull call to See ⟨undefined⟩ [scriba_NoRun()], page ⟨undefined⟩.

```
int scriba_SetVariable(pSbProgram pProgram,
                       long lSerial,
                       int type,
                       long lSetValue,
                       double dSetValue,
                       char *pszSetValue,
                       unsigned long size
);
```

The argument `lSerial` should be the serial number of the global variable as returned by See ⟨undefined⟩ [scriba_LookupVariableByName()], page ⟨undefined⟩.

The argument `type` should be one of the followings:

`SBT_UNDEF`

`SBT_DOUBLE`

`SBT_LONG`

`SBT_STRING`

`SBT_ZCHAR`

The function uses one of the arguments `lSetValue`, `dSetValue` or `pszSetValue` and the other two are ignored based on the value of the argument `type`.

If the value of the argument `type` is `SBT_UNDEF` all initialization arguments are ignored and the global variable will get the value `undef`.

If the value of the argument `type` is `SBT_DOUBLE` the argument `dSetValue` will be used and the global variable will be double holding the value.

If the value of the argument `type` is `SBT_LONG` the argument `lSetValue` will be used and the global variable will be long holding the value.

If the value of the argument `type` is `SBT_STRING` the argument `pszSetValue` will be used and the global variable will be long holding the value. The length of the string should in this case be specified by the variable `size`.

If the value of the argument `type` is `SBT_ZCHAR` the argument `pszSetValue` will be used and the global variable will be long holding the value. The length of the string is automatically calculated and the value passed in the variable `size` is ignored. In this case the string `pszSetValue` should be zero character terminated.

The funtion returns `SCRIBA_ERROR_SUCCESS` on success,

`SCRIBA_ERROR_MEMORY_LOW` if the data cannot be created or

`SCRIBA_ERROR_FAIL` if the parameter `lSerial` is invalid.

1.1.66 scriba_InitStaticModules()

This function calls the initialization functions of the modules that are statically linked into the interpreter. This is essential to call this fucntion from the embedding `main()` program in a variation that has one or more external modules staticallyl linked. If this function is not called the module initialization will not be called, because the module is never actually loaded and thus the operating system does not call the `DllMain` or `_init` function.

The function has to be called before the first interpreter thread starts. In case of a single thread variation this means that the function has to be called before the BASIC program starts.

The function does not take any argument and does not return any value.

```
void scriba_InitStaticModules(void
);
```

1.1.67 scriba_FinishStaticModules()

This function calls the finalization functions of the modules that are statically linked to the interpreter. Such a function for a dynamically loaded module is started by the operating system. Because the sttaically linked modules are not loaded the `_fini` function is not called by the UNIX loader and similarly the function `DllMain` is not called by Windows NT. Because some modules depend on the execution of this function this function has to be called after the last interpreter thread has finished.

```
void scriba_FinishStaticModules(void
);
```

1.2 basext.c

1.2.1 basext_GetArgsF()

This function can be used to get arguments simple and fast in extension modules. All functionality of this function can be individually programmed using the `besXXX` macros. Here it is to ease the programming of extension modules for most of the cases.

This function should be called like

```
iError = besGETARGS "ldz",&l1,&d1,&s besGETARGE
```

The macro `besGETARGS` (read GET ARGument Start) hides the complexity of the function call and the macro `besGETARGE` (read Get ARGument End) simply closes the function call.

The first argument is format string. Each character specifies how the next argument should be treated.

```
int basext_GetArgsF(pSupportTable pSt,
                    pFixSizeMemoryObject pParameters,
                    char *pszFormat,
```

```
                    ...
);
```

The following characters are recognized:

`i` the next argument of the function call should point to a `long` variable. The ScriptBasic argument will be converted to `long` using the macro `besCONVERT2LONG` and will be stored in the `long` variable.

`r` the same as `l` except that the argument should point a `double` and the basic argument is converted to `double` using `besCONVERT2DOUBLE`.

`z` the next argument should point to a `char *` pointer. The function takes the next BASIC argument as string, converts it to zero terminated string allocating space for it. These variables SHOULD be released by the caller using the macro `besFREE`.

`s` the next argument should point to a `unsigned char *` pointer. The function takes the next BASIC argument as string, converting it in case conversion is needed, and sets the `unsigned char *` pointer to point to the string. This format character should be used together with the character `l`

`l` the next argument should point to a `long` and the value of the variable will be the length of the last string atgument (either `z` or `s`). If there was no previous string argument the value returned will be zero.

`p` the next argument should point to a `void *` pointer. The BASIC argument value should be a string of `sizeof(void *)` characters that will be copied into the pointer argument. If the argument is not string or has not the proper size the function returns `COMMAND_ERROR_ARGUMENT_RANGE`.

`[` The arguments following this character are optional. Optional arguments may be unspecified. This is the case when the BASIC function call has less number of arguments or when the actual argument value is `undef`. In case of optional arguments the `undef` values are converted to zero value of the appropriate type. This means 0 in case of long, 0.0 in case of double, NULL in case of pointer and zero length string in case of strings.

`]` Arguments following this character are mandatory (are not optional). When the function starts to process the arguments they are mandatory by default. Using this notation you can enclode the optional arguments between `[` and `]`. For example the format string `"ii[z]"` means two long arguments and an optional zero terminated string argument.

`*` The argument is skipped. This may be used during development of a function.

The return value of the function is zero in case there is no error or the error code.

1.3 memory.c

1.3.1 memory_InitStructure()

Each execution context should have its own memory object responsible for the administration of the variables and the memory storing the values assigned to variables.

This function initializes such a memory object.

```
int memory_InitStructure(pMemoryObject pMo
);
```

1.3.2 memory_RegisterType()

This function should be used to register a variable type. The return value is the serial number of the type that should later be used to reference the type.

```
int memory_RegisterType(pMemoryObject pMo,
                        unsigned long SizeOfThisType
);
```

The argument of the function is the size of the type in terms of bytes. Usually this is calculated using the C structure `sizeof`.

If the type can not be registered -1 is returned.

1.3.3 memory_RegisterTypes()

This function should be used to initialize the usual `FixSizeMemoryObject` types. This sets some usual string sizes, but the caller may not call this function and set different size objects.

```
void memory_RegisterTypes(pMemoryObject pMo
);
```

This function registers the different string sizes. In the current implementation a string has at least 32 characters. If this is longer that that (including the terminating zchar) then a 64 byte fix size object is allocated. If this is small enough then a 128 byte fix size memory object is allocated and so on up to 1024 bytes. If a string is longer that that then a LARGE_OBJECT_TYPE is allocated.

The reason to register these types is that this memory management module keeps a list for these memory pieces and when a new short string is needed it may be available already without calling `malloc`. On the other hand when a `LARGE_OBJECT_TYPE` value is released it is always passed back to the operating system calling `free`.

1.3.4 memory_DebugDump()

This is a debugging function that dumps several variable data to the standard output. The actual behavior of the function may change according to the actual debug needs.

```
void memory_DebugDump(pMemoryObject pMo
);
```

1.3.5 memory_NewVariable()

This function should be used whenever a new variable is to be allocated. The function returns a pointer to a `FixSizeMemoryObject` structure that holds the variable information and pointer to the memory that stores the actual value for the memory.

If there is not engough memory or the calling is illegal the returned value is `NULL`

```
pFixSizeMemoryObject memory_NewVariable(pMemoryObject pMo,
                                        int type,
                                        unsigned long LargeBlockSize
);
```

The second argument gives the type of the memory object to be allocated. If this value is `LARGE_BLOCK_TYPE` then the third argument is used to determine the size of the memory to be allocated. If the type if NOT `LARGE_BLOCK_TYPE` then this argument is ignored and the proper size is allocated.

If the type has memory that was earlier allocated and released it is stored in a free list and is reused.

1.3.6 memory_ReleaseVariable()

This function should be used to release a memory object.

```
int memory_ReleaseVariable(pMemoryObject pMo,
                           pFixSizeMemoryObject p
);
```

1.3.7 memory_NewString()

This function should be used to allocate string variable.

```
pFixSizeMemoryObject memory_NewString(pMemoryObject pMo,
                                      unsigned long StringSize
);
```

The second argument specifies the length of th required string including.

The function checks the desired length and if this is small then is allocates a fix size object. If this is too large then it allocates a `LARGE_BLOCK_TYPE`

1.3.8 memory_NewCString()

This function should be used to allocate variable to store a constant string.

```
pFixSizeMemoryObject memory_NewCString(pMemoryObject pMo,
                                       unsigned long StringSize
);
```

The second argument specifies the length of the required string.

1.3.9 memory_SetRef()

Set the variable `ppVar` to reference the variable `ppVal`.

```
int memory_SetRef(pMemoryObject pMo,
                  pFixSizeMemoryObject *ppVar,
                  pFixSizeMemoryObject *ppVal
);
```

1.3.10 memory_NewRef()

```
pFixSizeMemoryObject memory_NewRef(pMemoryObject pMo
);
```

1.3.11 memory_IsUndef()

This function returns if the examined variable is `undef`. Since a variable containing `undef` but having other variables referencing this variable is NOT stored as `NULL` examining the variable agains `NULL` is not enough anymore since reference variables were introduced.

```
int memory_IsUndef(pFixSizeMemoryObject pVar
);
```

1.3.12 memory_Type()

This function returns the type of the variable. In case the program does not want to check the `NULL` undef, but wants to get `VTYPE_UNDEF` even if the variable is real `undef` being `NULL` calling this function is safe. Use this function instead of the macro `TYPE` defined in `command.h` is there is doubt.

```
int memory_Type(pFixSizeMemoryObject pVar
);
```

1.3.13 memory_SelfOrRealUndef()

```
pFixSizeMemoryObject memory_SelfOrRealUndef(pFixSizeMemoryObject pVar
);
```

1.3.14 memory_NewUndef()

```
pFixSizeMemoryObject memory_NewUndef(pMemoryObject pMo
);
```

1.3.15 memory_ReplaceVariable()

```
int memory_ReplaceVariable(pMemoryObject pMo,
                           pFixSizeMemoryObject *Lval,
                           pFixSizeMemoryObject NewValue,
                           pMortalList pMortal,
                           int iDupFlag
);
```

1.3.16 memory_NewLong()

```
pFixSizeMemoryObject memory_NewLong(pMemoryObject pMo
);
```

1.3.17 memory_NewDouble()

```
pFixSizeMemoryObject memory_NewDouble(pMemoryObject pMo
);
```

1.3.18 memory_CopyArray

```
pFixSizeMemoryObject memory_CopyArray(pMemoryObject pMo,
                                      pFixSizeMemoryObject p
);
```

1.3.19 memory_NewArray()

This function should be used whenever a new array is to be allocated.

```
pFixSizeMemoryObject memory_NewArray(pMemoryObject pMo,
                                     long LowIndex,
                                     long HighIndex
);
```

The index variables define the indices that are to be used when accessing an array element. The index values are inclusive.

1.3.20 memory_ReDimArray()

This function should be used when an array needs redimensioning. If the redimensioning is succesful the function returns the pointer to the argument `p`. If memory allocation is needed and the memory allocation fails the function returns `NULL`. In this case the original array is not changed.

If the redimensioned array is smaller that the original no memory allocation takes place, only the array elements (pointers) are moved.

```
pFixSizeMemoryObject memory_ReDimArray(pMemoryObject pMo,
                                       pFixSizeMemoryObject p,
                                       long LowIndex,
                                       long HighIndex
);
```

1.3.21 memory_CheckArrayIndex()

This function should be called before accessing a certain element of an array. The function checks that the index is within the index limitsof the array and in case the index is outside the index limits of the array it redimensionate the array.

The function returns the pointer passed as parameter `p` or NULL in case there is a memory allocation error.

```
pFixSizeMemoryObject memory_CheckArrayIndex(pMemoryObject pMo,
                                            pFixSizeMemoryObject p,
                                            long Index
);
```

1.3.22 memory_Mortalize()

This function should be used when a variable is to be put in a mortal list.

```
void memory_Mortalize(pFixSizeMemoryObject p,
                      pMortalList pMortal
);
```

Note that care should be taken to be sure that the variable is NOT on a mortal list. If the variable is already on a mortal list calling this function will break the original list and therefore may loose the variables that follow this one.

1.3.23 memory_Immortalize()

Use this function to immortalize a variable. This can be used when the result of an expression evaluation gets into a mortal variable and instead of copiing the value from the mortal variable to an immortal variable the caller can immortalize the variable. However it should know which mortal list the variable is on.

```
void memory_Immortalize(pFixSizeMemoryObject p,
                        pMortalList pMortal
);
```

1.3.24 memory_NewMortal()

When an expression is evaluated mortal variables are needed to store the intermediate results. These variables are called mortal variables. Such a variable is is allocated using this function and specifying a variable of type `MortalList` to assign the mortal to the list of mortal variables.

When the expression is evaluated all mortal variables are to be released and they are calling the function `memory_ReleaseMortals` (see See ⟨undefined⟩ [memory_ReleaseMortals()], page ⟨undefined⟩).

```
pFixSizeMemoryObject memory_NewMortal(pMemoryObject pMo,
                                      BYTE type,
                                      unsigned long LargeBlockSize,
                                      pMortalList pMortal
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.3.25 memory_DupImmortal()

This function creates a new mortal and copies the argument `pVar` into this new mortal.

```
pFixSizeMemoryObject memory_DupImmortal(pMemoryObject pMo,
                                        pFixSizeMemoryObject pVar,
                                        int *piErrorCode
);
```

1.3.26 memory_DupVar()

This function creates a new mortal and copies the argument `pVar` into this new mortal.

```
pFixSizeMemoryObject memory_DupVar(pMemoryObject pMo,
                                   pFixSizeMemoryObject pVar,
                                   pMortalList pMyMortal,
                                   int *piErrorCode
);
```

This function is vital, when used in operations that convert the values to `long` or `double`. Expression evaluation may return an immortal value, when the expression is a simple variable access. Conversion of the result would modify the value of the variable itself. Therefore functions and operators call this function to duplicate the result to be sure that the value they convert is mortal and to be sure they do not change the value of a variable when they are not supposed to.

Note that you can duplicate `long`, `double` and `string` values, but you can not duplicate arrays! The string value is duplicated and the characters are copied to the new location. This is perfect. However if you do the same with an array the array pointers will point to the same variables, which are not going to be duplicated. This result multiple reference to a single value. This situation is currently not supported by this system as we do not have either garbage collection or any other solution to support such memory structures.

1.3.27 memory_DupMortalize()

This function creates a new mortal and copies the argument `pVar` into this new mortal only if the value is immortal. If the value is mortal the it returns the original value.

```
pFixSizeMemoryObject memory_DupMortalize(pMemoryObject pMo,
                                         pFixSizeMemoryObject pVar,
                                         pMortalList pMyMortal,
                                         int *piErrorCode
);
```

1.3.28 memory_ReleaseMortals()

This function should be used to release the mortal variables.

When an expression is evaluated mortal variables are needed to store the intermediate results. These variables are called mortal variables. Such a variable is is allocated using this function and specifying a variable of type `MortalList` to assign the mortal to the list of mortal variables.

```
void memory_ReleaseMortals(pMemoryObject pMo,
                           pMortalList pMortal
);
```

1.3.29 memory_DebugDumpVariable()

This function is used for debugging purposes. (To debug ScriptBasic and not to debug a BASIC program using ScriptBasic. `:-o`)

The function prints the content of a variable to the standard output.

```
void memory_DebugDumpVariable(pMemoryObject pMo,
                              pFixSizeMemoryObject pVar
);
```

1.3.30 memory_DebugDumpMortals()

This function is used for debugging purposes. (To debug ScriptBasic and not to debug a BASIC program using ScriptBasic. `:-o`)

The function prints the content of the mortal list to the standard output.

```
void memory_DebugDumpMortals(pMemoryObject pMo,
                             pMortalList pMortal
);
```

1.3.31 memory_NewMortalString()

```
pFixSizeMemoryObject memory_NewMortalString(pMemoryObject pMo,
                                            unsigned long StringSize,
                                            pMortalList pMortal
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.3.32 memory_NewMortalCString()

```
pFixSizeMemoryObject memory_NewMortalCString(pMemoryObject pMo,
                                             unsigned long StringSize,
                                             pMortalList pMortal
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.3.33 memory_NewMortalLong()

```
pFixSizeMemoryObject memory_NewMortalLong(pMemoryObject pMo,
                                          pMortalList pMortal
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.3.34 memory_NewMortalRef()

This function was never used. It was presented in the code to allow external modules to create mortal reference variables. However later I found that the variable structure design does not allow mortal reference variables and thus this function is nonsense.

Not to change the module interface defintion the function still exists but returns NULL, like if memory were exhausted.

```
pFixSizeMemoryObject memory_NewMortalRef(pMemoryObject pMo,
                                         pMortalList pMortal
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.3.35 memory_NewMortalDouble()

```
pFixSizeMemoryObject memory_NewMortalDouble(pMemoryObject pMo,
                                            pMortalList pMortal
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.3.36 memory_NewMortalArray()

```
pFixSizeMemoryObject memory_NewMortalArray(pMemoryObject pMo,
                                           pMortalList pMortal,
                                           long IndexLow,
                                           long IndexHigh
);
```

If the parameter `pMortal` is `NULL` the generated variable is not mortal.

1.4 epreproc.c

1.4.1 External preprocessor handling

This module starts the external preprocessors.

=toc

1.4.2 Execute external preprocessors

This function executes the external preprocessors that are needed to be executed either by the command line options or driven by the extensions.

The command line option preprocessors are executed as listed in the character array `ppszArgPreprocessor`. These preprocessors are checked to be run first.

If there is no preprocessors defined on the command line then the preprocessors defined in the config file for the extensions are executed. The input file name is also modified by the code. The input file name is modified so that it will contain the source code file name after the preprocessing.

The return value of the function is the error code. This is `PREPROC_ERROR_SUCCESS` if the preprocessing was successful. This value is zero. If the return value is positive this is one of the error codes defined in the file `errcodes.def` prefixed by `PREPROC_`.

```
int epreproc(ptConfigTree pCONF,
             char *pszInputFileName,
```

```
                char **pszOutputFileName,
                char **ppszArgPreprocessor,
                void *(*thismalloc)(unsigned int),
                void (*thisfree)(void *)
);
```

The first argument `pCONF` is the configuration data pointer which is passed to the configuration handling routines.

The second argument `pszInputFileName` is the pointer to the pointer to the input file name.

The third argument is an output variable. This will point to the output file name upon success or to NULL. If this variable is NULL then an error has occured or the file needed no preprocessing. The two cases can be separated based on the return value of the function. If the file needed preprocessing and the preprocessing was successfully executed then this variable will point to a ZCHAR string allocated via the function `thismalloc`. This is the responsibility of the caller to deallocate this memory space after use calling the function pointed by `thisfree`.

The fourth argument `ppszArgPreprocessor` is an array of preprocessors to be used on the input file. This array contains pointers that point to ZCHAR strings. The ZCHAR strings contain the symbolic names of the external preprocessors that are defined in the configuration file. The configuration file defines the actual executable for the preprocessor and the temporary directory where the preprocessed file is stored. The final element of this pointer array should be `NULL`. If the pointer `ppszArgPreprocessor` is `NULL` or the pointer array pointed by this contains only the terminating `NULL` pointer then the extensions of the file name are used to determine what preprocessors are to be applied. Preprocessors are applied from left to right order of the file extensions.

The arguments `thismalloc` and `thisfree` should point to `malloc` and `free` or to a similar functioning function pair. These functions will be used via the `myalloc.c` module and also to allocate the new `pszOutputFileName` string in case of success. This means that the caller should use the function pointed by `thisfree` to release the string pointed by `pszOutputFileName` after the function has returned.

1.5 ipreproc.c

1.5.1 Internal preprocessor handling

=abstract This module loads the internal preprocessors =end

=toc

1.5.2 Initialize the preprocessor structure

This function is called after the `PreprocObject` was allocated. It initializes the preprocessor handling structures.

```
void ipreproc_InitStructure(pPreprocObject pPre
);
```

1.5.3 Release all memories allocated by preprocessors

This fucntion is called from the module `scriba_*` to release all memory that was allocated by the preprocessors and were not released.

```
void ipreproc_PurgePreprocessorMemory(pPreprocObject pPre
);
```

1.5.4 Insert a new preprocessor into the preprocessor list

The preprocessors that are active are stored in a linked list. When there is any action that needs a preprocessor this list is used to invoke the preprocessors. The preprocessors are invoked in the order they were entered into the system. For example if there are two lines in the source code saying:

```
use pre1
use pre2
```

then the preprocessor `pre1` is loaded first and `pre2` is loaded afterwards. When a preprocessor is invoked the preprocesor `pre1` is called first and `pre2` is called on the result. This function allocates a list element and inserts it to the end of the list.

```
pPreprocessor ipreproc_InsertPreprocessor(pPreprocObject pPre
);
```

The argument is the preprocessor object, environment.

The return value is pointer to the list element or `NULL` if memory allocation occured.

1.5.5 Delete a preprocessor from the list of preprocessors

This function deletes a preprocessor from the list of preprocessors. The preprocessor was inserted into the list using the function `InsertPreprocessor`.

This function unhooks the element from the list and also releases the memory that was occupied by the list element. The function does not unload the shared object (or DLL under Windows NT) from the memory.

```
void ipreproc_DeletePreprocessor(pPreprocObject pPre,
                                 pPreprocessor pOld
);
```

The first argument is the preprocessor object environment. The second argument is the pointer to the list element to be deleted.

1.5.6 Load an internal preprocessor

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.

1.5.7 Process preprocessor requests

This function is used by ScriptBasic at certain points of the execution to start the preprocessors. It calls each loaded preprocessor one after the other until there is no more preprocessors or one of them alters the command variable to `PreprocessorDone`.

This function gets three arguments.

```
int ipreproc_Process(pPreprocObject pPre,
                     long lCommand,
                     void *pPointer
);
```

`pRe` is the preprocessor object.

`lCommand` is the command for the preprocessor to execute. For the possible values look at the file `prepext.h` (created from `prepext.c`) `enum PreprocessorCommands`

`pPointer` is a pointer to a structure. The structure actually depends on the actual value of `lCommand`. For different commands this pointer points to different structures.

When the preprocessors are called they can alter the `long` variable `lCommand` passed to them by reference.

When a preprocessor in this variable returns the value `PreprocessorDone` the preprocessing in the actual stage is stopped and no further proreprocessor is invoked. However this has no effect on later preprocessor incocation. Returning this value in this variable solely means that the preprocessor has done all work that has to be done at the very point and thus there is no need of further preprocessor handling.

When a preprocessor in this variable returns the value `PreprocessorUnload` the function unhooks the preprocessor from the list af active preprocessors, releases all memory that the preprocessor used up and frees the library.

The return value of the preprocessor functions should be zero or error code.

The return value of the function `ipreproc_Process` is zero or the error value of a preprocessor. If a preprocessor returns a non-zero error code no further preprocessors are invoked.

This function can not be used in situation when the preprocessors may return other value in `lCommand` than `PreprocessorDone` or `PreprocessorContinue`.

1.5.8 Preprocessor function

This function has to be implemented in the external preprocessor and exported from the DLL/SO file. It has to have the following prototype.

```
int DLL_EXPORT preproc(void *p, long *pFUN, void *q);
```

1.6 command.c

1.6.1 Header file for command building

This file contains macros that help the command implementators to write easy, readable and compact code. The macros on one hand assume some variable naming, but on the other hand hide some details of the function calls.

For examples how to use these macros see the files in the `commands` directory of the source files. Note that some command implementation do not use these macros, because they do sophisticated and low level operations that have to deal with the interpreted code in more detail. However such a programming should not be neccesary to implement a new command or function for extending the language.

To implement a new command or function do the following:

Read the definitions here.

Read the macros and see the implemented functions and commands in the directory `commands`. Try to get some understanding how it works.

Take an already implemented function which is similar to the one that you want to implement. Copy it to a new file, give it a new name.

Edit the `syntax.def` file to include the new command or file and run `syntaxer.pl syntax.def` to generate the files `syntax.h` and `syntax.def`

Compile ScriptBasic to see that the function with the new name or the command has the same functionality as the old one that you have copied.

Start modifying the code step by step. At each step compile the interpreter and check that the modified functionality exists.

Debug, crosscheck the final code and document your new command or function for your project.

Announce the new functionality in your project and be proud.

Be happy.

NOTE:

This file is actually a header file. This is maintained in `command.c` to avoid accidental deletion of the `command.h` file. The file `command.h` is an intermediate file created from `command.c` using the Perl utility `headerer.pl`. Because all `*.h` files are intermediate it would have been dangerous to have `command.h` to be a source file.

The macros and types defined in this file:

1.6.2 Start a command implementation

```
COMMAND
```

This macro should be used to start a function that implements a command or built-in function. This actually generates the function header with some local variable declarations and some variable setting.

```
COMMAND(FUNCTIONNAME)
```

in the current implementation generates:

```
void COMMAND_FUNCTIONNAME(pExecuteObject pEo)
  MortalList _ThisCommandMortals=NULL;
  pMortalList _pThisCommandMortals = &_ThisCommandMortals;
  unsigned long _ActualNode=PROGRAMCOUNTER;
  int iErrorCode;
```

Note that further implemenation changes may change the actual code generated not followed in this documentation. However the actual use of the macro should not change.

The function should be finished using the macro See ⟨undefined⟩ [END], page ⟨undefined⟩ documented also in this documentation.

1.6.3 Finish a command implementation

This macro generates the finishing code that a function impementing a BASIC command or built-in function should have.

```
END
```

in the current implementation generates the following code:

```
goto _FunctionFinishLabel;
_FunctionFinishLabel:
memory_ReleaseMortals(pEo->pMo,&_ThisCommandMortals);
iErrorCode = 0;
```

Note that further implemenation changes may change the actual code generated not followed in this documentation. However the actual use of the macro should not change.

Some part of the code may seem unneccesary. The `goto` just before the label, or the final assignment to a local variable. These are present to avoid some compiler warning and clever compilers should optimize out these constructs.

1.6.4 Implement a command that has identical functionality

This macro helps to implement a command that has identical functionality as another command. You can see examples for it in the looping construct. There is a wide variety of looping construct to implement all looping facilities that BASIC programmers got used to. However the loop closing commands more or less behave the same. For example the command `next` behaves exactly the same as the command `while`

Also note that the identical behaviour does not mean that one command can be used instead of the other. There are conventions that the BASIC language requires and the syntactical analyzer does nto allow to close a `FOR` loop using a `WEND` command.

To present an example on how to use this macro we have copied the code of the comman `NEXT`:

```
COMMAND(NEXT)

  IDENTICAL_COMMAND(WEND)

END
```

1.6.5 Use the mortals of the caller

```
USE_CALLER_MORTALS
```

You should use this macro when impementing a built-in function. The implementation of the commands use their own mortal list to collect mortal variables storing intermediate results. Built-in function implementations do NOT maintain their own collection of mortal variables. This macro sets some variables to collect mortal variables into the list of the calling modules.

To get a deeper understanding of mortals and variable handling see the documentation for the source file `memory.c`

1.6.6 Return from the function

```
RETURN
```

When implementing a built-in function or command you should never ever `return` from the function because that may avoid release of mortal variables and may not execute the final code which is needed to properly finish the function. Use the macro `RETURN` instead.

1.6.7 Terminate a function with error

```
ERROR(x)
```

Use this macro to terminate the execution of a commans or built-in function with some error. `ERROR(0)` means no error, but this construct is not advisable, use See ⟨undefined⟩ [RETURN], page ⟨undefined⟩ instead. Any other code value can be used to specify a special error.

1.6.8 The value of the programcounter

```
PROGRAMCOUNTER
```

This macro results the node id of the command, which is currently executed. Note that this is already the node that contains the command code and not the code that the class variable `ProgramCounter` points. `ProgramCounter` points to a list node. This list node points to the node returned by `PROGRAMCOUNTER` and to the next command node.

1.6.9 Implement jump instructions

```
SETPROGRAMCOUNTER(x)
```

Use this macro when a command decides that the code interpretation should continue at a different location. The simplest example on how to use this macro is the implementation of the command `goto`:

```
COMMAND(GOTO)

  SETPROGRAMCOUNTER(PARAMETERNODE);

END
```

See also See ⟨undefined⟩ [PARAMETERNODE], page ⟨undefined⟩.

1.6.10 Get the next command parameter

```
NEXTPARAMETER
```

This macro should be used to get access to the next command parameter. This macro should NOT be used in built-in function implemenation. The functions have only a single parameter, which is indeed an expression list. To access the parameters of a function use the macros See ⟨undefined⟩ [PARAMETERLIST], page ⟨undefined⟩, See ⟨undefined⟩ [CAR], page ⟨undefined⟩ and See ⟨undefined⟩ [CDR], page ⟨undefined⟩.

When you implement a command you can get the first parameter of a command using the macro `PARAMETERNODE`, `PARAMETERLONG`, `PARAMETERDOUBLE` or `PARAMETERSTRING` (see See ⟨undefined⟩ [PARAMETERXXX], page ⟨undefined⟩). If the command has more than one parameters you should use the macro `NEXTPARAMETER` to step to the next parameter.

1.6.11 Access a command parameter

```
PARAMETERNODE
PARAMETERLONG
PARAMETERDOUBLE
PARAMETERSTRING
```

You should use these macros to get access to the command parameters. Usually these parameters are presented as "nodes". Syntax definition usually allows you to use expressions whenever a long, double or string is expected. Expressions are converted to "nodes" and therefore only a few commands may use the macros `PARAMETERLONG`, `PARAMETERDOUBLE` or `PARAMETERSTRING`. These can be used when the parameter is a long number, double number or a contant string and NOT an expression or expression list.

When a command has more than one parameters you can access each, step by step using the macro See ⟨undefined⟩ [NEXTPARAMETER], page ⟨undefined⟩, which steps onto the next parameter of the command.

Do **NOT** use any of the `PARAMETERXXX` macro or the macro See ⟨undefined⟩ [NEXTPARAMETER], page ⟨undefined⟩ in the implementation of a built-in function.

1.6.12 Get the opcode of a node

```
OPCODE(x)
```

This macro results the opcode of the node `x`. This macro can rarely be used by your extension.

1.6.13 Get the parameter list node for a function

```
PARAMETERLIST
```

You should use this macro to get the parameter list for a built-in function or operator. Both built-in functions and operators get their parameter list as and expression list. This macro gets the first list node of the expression list.

The parameter is presented as an expression list even if there is only a single parameter for the function.

To access the parameters use the macros See ⟨undefined⟩ [CAR], page ⟨undefined⟩ and See ⟨undefined⟩ [CDR], page ⟨undefined⟩.

1.6.14 Get the car node of a list node

Expression lists and commands are stored using list nodes. A list node has an See ⟨undefined⟩ [OPCODE], page ⟨undefined⟩ value `eNTYPE_LST` defined in `expression.c`, and has two node pointers. One points to the node that belongs to the list member and other points to the next list node.

If `nItem` is a list node for an expression list then `CAR(nItem)` is the root node of the expression, and `CDR(nItem)` is the list node for the next expression. `CAR(CDR(nItem))` is the root node of the second expression.

The nodes are indexed with `unsigned long` values. `NULL` pointer is a `0L` value and list node lists are terminated with a node that has `CDR(nItem)=0L`.

See also See ⟨undefined⟩ [CDR], page ⟨undefined⟩.

1.6.15 Get the cdr node of a list node

Expression lists and commands are stored using list nodes. A list node has an See ⟨undefined⟩ [OPCODE], page ⟨undefined⟩ value `eNTYPE_LST` defined in `expression.c`, and has two node pointers. One points to the node that belongs to the list member and other points to the next list node.

If `nItem` is a list node for an expression list then `CAR(nItem)` is the root node of the expression, and `CDR(nItem)` is the list node for the next expression. `CAR(CDR(nItem))` is the root node of the second expression.

The nodes are indexed with `unsigned long` values. `NULL` pointer is a `0L` value and list node lists are terminated with a node that has `CDR(nItem)=0L`.

See also See ⟨undefined⟩ [CAR], page ⟨undefined⟩.

1.6.16 Special variable to store the result

```
RESULT
```

Use this macro to store the result of the operation. Usually a new mortal value should be allocated using See ⟨undefined⟩ [NEWMORTALXXX], page ⟨undefined⟩ and the appropriate value of `RESULT` should be then set.

See also See ⟨undefined⟩ [NEWMORTALXXX], page ⟨undefined⟩, See ⟨undefined⟩ [XXXVALUE], page ⟨undefined⟩

1.6.17 Access certain values of a memory object

```
STRINGVALUE(x)
LONGVALUE(x)
DOUBLEVALUE(x)
```

These macros are actually defined in `memory.c`, but we document them here because they play an important role when writing implementation code for functions and operators.

These macros get the string (car*), long or double value of a variable. The macros can also be used to assign value a long or double value to a variable. Do not forget to change the type of the variable. You usually should call the macro See ⟨undefined⟩ [CONVERT2XXX], page ⟨undefined⟩.

Note that you should NOT change the string value of a variable. The `STRINGVALUE(x)` is a (char *) pointer to a string. You have to change the characters in this string, or you should allocate a new string with longer or shorter length and copy the characters, but never change the (char *) pointer.

1.6.18 Create a new mortal value

```
NEWMORTALLONG
NEWMORTALDOUBLE
NEWMORTALSTRING(length)
```

Use these macros to allocate a new mortal variable. In case of a string you have to give the length of the string.

=center INCLUDE THE TERMINATING ZERO IN THE LENGTH!!! =nocenter

Never allocate non-mortal values when implemenating operators or functions.

1.6.19 Evaluate an expression

```
EVALUATEEXPRESSION(x)
_EVALUATEEXPRESSION(x)
```

Use these macros to evaluate an expression. The argument is the root node of the expression. When a command has a parameter, which is an expression you should write:

```
VARIABLE Param;

Param = EVALUATEEXPRESSION(PARAMETERNODE);
```

Implementing a function or operator you should write

```
VARIABLE Param;

Param = EVALUATEEXPRESSION(CAR(PARAMETERLIST));
```

For further details see examples in the source files `commands/let.c`, `commands/mathops.c`.

NOTE:

When an expression is evaluated the returned pointer points to a struct which contains the value of the expression. This is usually a mortal variable which was created during expression evaluation. However in some cases this pointer points to a nonmortal variable. If the expression is a single global or local variable the result of the expression is the pointer to the variable value.

This helps writing for more efficient code. On ther other hand operators tend to convert the operands to long or double in case they expect a long or double but get a different type. The conversions, or other manipulations then change the original variable value, which is a side effect. For this reason the macro `EVALUATEEXPRESSION` also calls `memory_DupMortalize` which creates a new variable, and copies the content of the variable passed as argument if the variable is not a mortal.

`_EVALUATEEXPRESSION` does not do this and therefore you can use it for more efficient memory handling avoiding the creation of unneccesary copies of variables.

If you are not sure whichone to use use the first one without the leading underscore.

1.6.20 Evaluate a left value

```
EVALUATELEFTVALUE(x)
```

Use this macro to evaluate a left value. This is done in the function `commands/let.c` that implements the command LET. This command wasprogrammed with comments to help the learning how this works.

1.6.21 Immortalize a variable

```
IMMORTALIZE(x)
```

Use this macro to immortalize a variable.You usually should avoid immortalizing values when implementing operators of functions. Immortalize a value when it is assigned to a variable.

1.6.22 Create a new immortal value

```
NEWLONG
NEWDOUBLE
NEWSTRING(length)
```

Use these macros to create a new long, double or string value. The created value is NOT mortal. You usually should avoid this when implementing functions or operators. However you may need in some command implementation.

1.6.23 Convert a value to other type

```
CONVERT2DOUBLE(x)
CONVERT2LONG(x)
CONVERT2STRING(x)
```

Use these macros to convert a value to long, double or string. The macros return the pointer to the converted type. Note that conversion between long and double does not generate a new value. In such a conversion the argument pointer is returned and the value itself is converted from one type tp the other. This was programmed this way to avoid unneccesary creation of mortal values. However be sure that either the argument is a mortal value or you do not mind the conversion of the value and so the value of the variable that the value was assigned to. You need not worry about this when you use the macro `EVALUATEEXPRESSION` and not `_EVALUATEEXPRESSION`.

Also note that a conversion does not duplicate the value if the value already has the desired type. In such a case the argument pointer is returned.

On the other hand there is no guarantee that the conversion is done in-place. When conversion from string to anything else is done a new mortal variable is allocated and the pointer to that value is returned.

1.6.24 Parameter pointer

```
PARAMPTR(x)
THISPARAMPTR
```

Each command may need some parameters that are persistent during a program execution. For example the file handling routines need an array that associates the opened file handles with the integer values that the basic language uses. Each command may allocate storage and assign a pointer to `THISPARAMPTR` to point to the allocated space. This is a void pointer initialized to `NULL`.

A command may access a pointer of another command using the macro `PARAMPTR(x)` supplying x as the command code. This is usually `CMD_XXX` with XXX the name of the command, function or operator.

See also See ⟨undefined⟩ [ALLOC], page ⟨undefined⟩ and See ⟨undefined⟩ [FREE], page ⟨undefined⟩

1.6.25 Allocate memory

The `ALLOC` and See ⟨undefined⟩ [FREE], page ⟨undefined⟩ macros are provided to allocate general memory. They are not intended to create new variable storage. For that purpose the See ⟨undefined⟩ [NEWXXX], page ⟨undefined⟩ and See ⟨undefined⟩ [NEWMORTALXXX], page ⟨undefined⟩ macros should be used.

The memory allocated should usually be assigned to the `THISPARAMPTR` pointer (see See ⟨undefined⟩ [PARAMPTR], page ⟨undefined⟩).

The macro `ALLOC` behaves similar to the system function `malloc` accepting the size of the required memory in bytes and returning a void pointer.

The macro See ⟨undefined⟩ [FREE], page ⟨undefined⟩ accepts the pointer to the allocated memory and returns nothing.

There is no need to release the allocated memory. The memory allocated using `ALLOC` is automatically release upon program termination.

1.6.26 Release memory

The See ⟨undefined⟩ [ALLOC], page ⟨undefined⟩ and `FREE` macros are provided to allocate general memory. They are not intended to create new variable storage. For that purpose the See ⟨undefined⟩ [NEWXXX], page ⟨undefined⟩ and See ⟨undefined⟩ [NEWMORTALXXX], page ⟨undefined⟩ macros should be used.

The memory allocated should usually be assigned to the `THISPARAMPTR` pointer (see See ⟨undefined⟩ [PARAMPTR], page ⟨undefined⟩).

The macro See ⟨undefined⟩ [ALLOC], page ⟨undefined⟩ behaves similar to the system function `malloc` accepting the size of the required memory in bytes and returning a void pointer.

The macro `FREE` accepts the pointer to the allocated memory and returns nothing.

There is no need to release the allocated memory. The memory allocated using See ⟨undefined⟩ [ALLOC], page ⟨undefined⟩ is automatically release upon program termination.

1.6.27 Decide if a string is integer or not

```
ISSTRINGINTEGER(x)
```

Use this macro to decide that a string contains integer value or not. This can beuseful implementing operators that work with doubles as well as longs. When one of the operators is a string hopefully containing the decimal form of a number you have to convert the string to a long or double. This macro calls a function that decides whether the string contains an integer number convertible to long or contains a double.

For an example how to use it see the source file `commands/mathops.c`

1.6.28 Basic C variable types to be used

```
NODE nNode;
VARIABLE Variable;
LEFTVALUE Lval;
```

These `typedef`s can be used to declare C variables that are to hold value associated with nodes, variables (or values) and left values. For example how to use these `typedef`s see the files `commands/let.c`, `commands/mathops.c`

1.6.29 Get the actual type of a value

```
TYPE(x)
```

Use this macro to access the type of a value. Values can hold long, double, string or reference types. This macro returns the type of the value. For comparision use the constants:

`VTYPE_LONG` long value

`VTYPE_DOUBLE` double value

`VTYPE_STRING` string value

`VTYPE_REF` reference value

1.7 lexer.c

This module contains the functions and structures that are used by ScriptBasic to perform lexical analysis of the source code. The module was originally developed for ScriptBasic but was developed to be general enough to be used in other projects.

1.7.1 lex_SymbolicName()

This function usually is for debug purposes. This searches the table of the predefined symbols and returns the string which is the predefined symbols for which the code was passsed.

```
char *lex_SymbolicName(pLexObject pLex,
                       long OpCode
);
```

1.7.2 lex_HandleContinuationLines()

This function is called from the main function before syntax analysis is started. This function handles the usual basic continuation lines. If the last character on a line is a _ character, which is either recognised during lexical analysis as a character or as a symbol then this lexical element and the following new-line character token is removed from the list of tokens.

```
void lex_HandleContinuationLines(pLexObject pLex
);
```

1.7.3 lex_RemoveSkipSymbols()

This function is called from See ⟨undefined⟩ [lex_DoLexicalAnalysis()], page ⟨undefined⟩ to remove the lexical elements from the list of tokens that were denoted by the preprocessors to be deleted.

Some lexical elements are used to give information to some of the preprocessors. These tokens should be deleted, because later processing can not deal with them and confuses syntax analysis.

In those cases the preprocessor should set the type of the token to be `LLEX_T_SKIP` or `LEX_T_SKIP_SYMBOL`. 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.

```
void lex_RemoveSkipSymbols(pLexObject pLex
);
```

1.7.4 lex_RemoveComments()

This function called from the function See ⟨undefined⟩ [lex_DoLexicalAnalysis()], page ⟨undefined⟩ function to remove the comments before the syntax analysis starts.

It should be called before calling the continuation line handling because usually REM lines are not continuable

```
void lex_RemoveComments(pLexObject pLex
);
```

1.7.5 lex_NextLexeme()

Use this function during iteration to get the next lexeme from the list of lexemes.

```
void lex_NextLexeme(pLexObject pLex
);
```

1.7.6 lex_SavePosition()

Use this function to save the current position of the iteration. This is neccessary during syntactical analysis to return to a certain position when syntactical analysis fails and the program has to go back and try a different command syntax.

```
void lex_SavePosition(pLexObject pLex,
                      pLexeme *ppPosition
);
```

The second argument is a `pLexeme *` type variable that holds the position and should be passed as argument to the function See ⟨undefined⟩ [lex_RestorePosition()], page ⟨undefined⟩.

1.7.7 lex_RestorePosition()

Use this function to restore the lexeme position that was saved calling the function See ⟨undefined⟩ [lex_SavePosition()], page ⟨undefined⟩

```
void lex_RestorePosition(pLexObject pLex,
                         pLexeme *ppPosition
);
```

1.7.8 lex_StartIteration()

You should call this function when the list of lexemes was built up before starting the iteration of the syntax analyzer. This function sets the iteration pointer to the first lexeme.

```
void lex_StartIteration(pLexObject pLex
);
```

1.7.9 lex_EOF()

Call this function to check if the iteration has reached the last lexeme.

```
int lex_EOF(pLexObject pLex
);
```

1.7.10 lex_Type()

During lexeme iteration this function can be used to retrieve the typeof the current lexeme. The type of a lexeme can be:

`LEX_T_DOUBLE` a double value. A number which is not integer.

`LEX_T_LONG` an long value. A number which is integer.

`LEX_T_STRING` a string.

`LEX_T_ASYMBOL` an alpha symbol, like a variable. This symbol is not predefined. The value of the lexeme is the string of the symbol.

`LEX_T_NSYMBOL` a predefined symbol. The actual value of the lexeme is the token value of the symbol. If you wan to get the actual string of the symbol you have to call the function See ⟨undefined⟩ [lex_SymbolicName()], page ⟨undefined⟩.

`LEX_T_CHARACTER` A character that is not a predefined symbol and does not fit into any string.

```
int lex_Type(pLexObject pLex
);
```

1.7.11 lex_Double()

When the type of the current lexeme is `LEX_T_DOUBLE` during the lexeme iteration this function should be used to retrieve the actual value of the current lexeme.

```
double lex_Double(pLexObject pLex
```

1.7.12 lex_String()

When the type of the current lexeme is `LEX_T_STRING` during the lexeme iteration this function should be used to retrieve the actual value of the current lexeme.

```
char *lex_String(pLexObject pLex
);
```

1.7.13 lex_StrLen()

When the type of the current lexeme is `LEX_T_STRING` during the lexeme iteration this function should be used to retrieve the length of the current lexeme. This is more accurate than calling `strlen` on the actual string because the string itself may contain zero characters.

```
long lex_StrLen(pLexObject pLex
);
```

1.7.14 lex_Long()

When the type of the current lexeme is `LEX_T_LONG` during the lexeme iteration this function should be used to retrieve the actual value of the current lexeme.

```
long lex_Long(pLexObject pLex
);
```

1.7.15 lex_LineNumber()

This function returns the line number that the actual lexeme is in the source file. This function is needed to print out syntax and lexical error messages.

See also See ⟨undefined⟩ [lex_FileName()], page ⟨undefined⟩.

```
long lex_LineNumber(pLexObject pLex
);
```

1.7.16 lex_FileName()

This function returns a pointer to a constant string which is the file name that the lexeme was read from. Use this function to print out error messages when syntax or lexical error occures.

See also See ⟨undefined⟩ [lex_LineNumber()], page ⟨undefined⟩.

```
char *lex_FileName(pLexObject pLex
);
```

1.7.17 lex_XXX()

These access functions are implemented as macros and are put into <lexer.h> by the program `headerer.pl`

The macros access `Int`, `Symbol`, `Float` etc values of the current lexeme. However these are strored in a location which is named a bit different. For example the string of a symbol is stored in the string field of the lexeme. To be readable and to be compatible with future versions use these macros to access lexeme values when lexeme has any of these types.

```
/*
TO_HEADER:
#define lex_Int(x) lex_Long(x)
#define lex_Symbol(x) lex_String(x)
```

```
#define lex_Float(x) lex_Double(x)
#define lex_Char(x) lex_Long(x)
#define lex_Token(x) lex_Long(x)
#define lex_Code(x) lex_Long(x)
*/
/*
```

1.7.18 lex_Finish()

Call this functionto release all memory allocated by the lexical analyzer.

```
void lex_Finish(pLexObject pLex
);
```

1.7.19 lex_DumpLexemes()

Us this function for debugging. This function dumps the list of lexemes to the file `psDump`.

```
void lex_DumpLexemes(pLexObject pLex,
                     FILE *psDump
);
```

1.7.20 lex_ReadInput()

Call this function after proper initialization to read the input file. This function performs the laxical analysis and builds up an internal linked list that contains the lexemes.

```
int lex_ReadInput(pLexObject pLex
);
```

1.7.21 lex_InitStructure()

You may but need not call this function to initialize a `LexObject`. You may also call this function to use the settings of the function and set some variables to different values after the function returns.

```
void lex_InitStructure(pLexObject pLex
);
```

1.8 expression.c

The functions in this file compile a ScriptBasic expression into internal form. The functions are quite general, and do NOT depend on the actual operators that are implemented in the actual version.

This means that you can define extra operators as well as extra built-in functions easily adding entries into tables and need not modify the compiling code.

1.8.1 What is an expression in ScriptBasic

Altough the syntax defintion in script basic is table driven and can easily be modified expressions are not. The syntax of an expression is somewhat fix. Here we formally define what the program thinks to be an expression. This restriction should not cause problem in the usage of this module because this is the usual syntax of an expression. Any altering to this would result in an expression syntax which is unusual, and therefore difficult to use for the common users. The operators and functions along with therir precedence values are defined in tables anyway, so you have flexibility.

The formal description of an expression syntax:

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

left_value ::= variable  '[' expression_list ']'
               variable '' expression_list '' .
```

UNOP

is unary operator as defined in tables in file operators.h

NUMBER

is a number, lexical element.

STRING

is a string, lexical element.

VARIABLE

is a lexical element.

FUNC

is a function either built in, or user defined

OP(i)

is an operator of precendece i as defined in tables.

1.8.2 ex_DumpVariables()

This function dumps the variables stored in the symbol table to the file pointed by `fp`

```
void ex_DumpVariables(SymbolTable q,
                      FILE *fp
);
```

Note that this function is a debug function.

1.8.3 expression_PushNameSpace()

When a `module name` instruction is encountered the name space is modified. However the old name space should be reset when an `end module` statement is reached. As the modules can be nested into each other the name spaces are stored in a name space stack during syntax analysis.

This function pushes the current name space onto the stack. After calling this function the caller can put the new string into the `pEx->CurrentNameSpace` variable and later calling See ⟨undefined⟩ [ex_PopNameSpace()], page ⟨undefined⟩ can be called to retrive the saved name space.

```
int expression_PushNameSpace(peXobject pEx
);
```

1.8.4 ex_CheckUndefinedLabels()

This function traverses the label symbol table and reports all undefined labels as error. Undefined labels reference the node with node-number zero. Jumping on a label like that caused the program to stop instead of compile time error in previous versions.

```
void ex_CheckUndefinedLabels(peXobject pEx
);
```

1.8.5 ex_CleanNameSpaceStack()

This function cleans the name space stack. This cleaning does not need to be done during syntax analysis. It is needed after the analysis has been done to detect unclosed modules.

Note that the `main::` module is implicit and can not and should not be closed unless it was explicitly opened.

The function calls the report function if the name space is not empty when the function is called.

```
void ex_CleanNameSpaceStack(peXobject pEx
);
```

1.8.6 expression_PopNameSpace()

When a `module name` instruction is encountered the name space is modified. However the old name space should be reset when an `end module` statement is reached. As the modules can be nested into each other the name spaces are stored in a name space stack during syntax analysis.

This function pops the name space from the name space stack and copies the value to the `pEx->CurrentNameSpace` variable. This should be executed when a name space is closed and we want to return to the embedding name space.

```
int expression_PopNameSpace(peXobject pEx
);
```

1.8.7 ex_PushWaitingLabel()

This function is used to define a label.

```
int ex_PushWaitingLabel(peXobject pEx,
                        pSymbolLABEL pLbl
);
```

When a label is defined the `eNode_l` that the label is going to belong still does not exists, and therefore the `NodeId` of that `eNode_l` is not known. This function together with See ⟨undefined⟩ [ex_PopWaitingLabel()], page ⟨undefined⟩ maintains a stack that can store labels which are currently defined and still need a line to be assigned to them. These labels all point to the same line. Altough it is very rare that many labels point to the same line, it is possible. The number of labels that can point the same line is defined by the constant `MAX_SAME_LABELS` defined in `expression.c`

To make it clear see the following BASIC code:

```
this_is_a_label:
REM this is a comment
            PRINT "hello word!!"
```

The label is defined on the first line of the example. However the label belongs to the third line containing the statement `PRINT`. When the label is processed the compiler does not know the node number of the code segment which is generated from the third line. Therefore this function maintains a label-stack to store all labels that need a line. Whenever a line is compiled so that a label can be assigned to that very line the stack is emptied and all labels waiting on the stack are assigned to the line just built up. (Or the line is assigned to the labels if you like the sentence the other way around.)

Note that not only labels given by a label defining statement are pushed on this stack, but also labels generated by commands like 'while/wend' of 'if/else/endif'.

1.8.8 ex_PopWaitingLabel()

This function is used to get a label out of the waiting-label-stack.

```
pSymbolLABEL ex_PopWaitingLabel(peXobject pEx
);
```

To get some description of waiting labels see the description of the function See ⟨undefined⟩ [ex_PushWaitingLabel()], page ⟨undefined⟩.

1.8.9 _ex_PushLabel()

This function is used to push an unnamed label on the compile time stack. For more detailed defintion of the unnamed labels and this stack see the documentation of the function See ⟨undefined⟩ [ex_PopLabel()], page ⟨undefined⟩.

```
int _ex_PushLabel(peXobject pEx,
                  pSymbolLABEL pLbl,
                  long Type,
                  void *pMemorySegment
);
```

The argument `Type` is used to define the type of the unnamed label. This is usually defined in the table created by the program `syntaxer.pl`

=bold Do NOT get confused! This stack is NOT the same as the waiting label stack. That is usually for named labels. =nobold

However the non-named labels are also pushed on that stack before they get value.

1.8.10 _ex_PopLabel()

This function is used to pop an unnamed label off the compile stack.

When a construct, like `IF/ELSE/ENDIF` or `REPEAT/UNTIL` or `WHILE/WEND` is created it is defined using compile time label stack.

For example analyzing the instruction `WHILE` pushes a "go forward" value on the compile time label stack. When the instruction `WEND` is analyzed it pops off the value and stores `NodeId` for the label. The label itself is not present in the global label symbol table, because it is an unnamed label and is referenced during compile time by the pointer to the label structure.

The value of the `AcceptedType` ensures that a `WEND` for example do not matches an `IF`.

```
pSymbolLABEL _ex_PopLabel(peXobject pEx,
                          long *pAcceptedType
);
```

The array `pAcceptedType` is an array of long values that have `MAX_GO_CONSTANTS` values. This is usually points to a static table element which is generated by the program `syntaxer.pl`.

=bold Do NOT get confused! This stack is NOT the same as the waiting label stack. That is for named labels. =nobold

1.8.11 _ex_CleanLabelStack()

This function is used to clean the unnamed label stack whenever a locality is left. This helps to detect when an instruction like `FOR` or `WHILE` is not closed within a function.

```
void _ex_CleanLabelStack(peXobject pEx
);
```

1.8.12 Some NOTE on SymbolXXX functions

The functions named `SymbolXXX` like `SymbolLABEL`, or `SymbolUF` do NOT store the names of the symbols. They are named `SymbolXXX` because they are natural extensions of the symbol table system. In other compilers the functionality to retrieve the arguments of a symbol is part of the symbol table handling routines.

In script basic the symbol table handling routines were developed to be general purpose. Therefore all the arguments the symbol table functions bind toa symbol is a `void *` pointer. This pointer points to a struct that holds the arguments of the symbols, and the functions `SymbolXXX` allocate the storage for the arguments.

This way it is possible to allocate arguments for non-existing symbols, as it is done for labels. Script basic uses non-named labels to arrange the "jump" instructions for `IF/ELSE/ENDIF` constructs. (And for some other constructs as well.) The label and jump constructs look like:

```
        IF expression Then

        ELSE
label1:


        END IF
label2:
```

The labels `label1` and `label2` do not have names in the system, not even autogenerated names. They are referenced via pointers and their value (the `NodeId` of the instruction) get into the `SymbolLABEL` structure and later int o the `cNODE` during build.

1.8.13 _new_SymbolLABEL()

This function should be used to create a new label. The label can be named or unnamed. Note that this structure does NOT contain the name of the label.

```
pSymbolLABEL _new_SymbolLABEL(peXobject pEx
);
```

Also note that all labels are global in a basic program and are subject to name space decoration. However the same named label can not be used in two different functions in the same name space.

A label has a serial value, which is not actually used and a number of the node that it points to.

See the comments on See ⟨undefined⟩ [ex_symbols()], page ⟨undefined⟩.

1.8.14 _new_SymbolVAR()

This function should be used to create a new variable during compile time. A variable is nothing else than a serial number. This serial number starts from 1.

```
pSymbolVAR _new_SymbolVAR(peXobject pEx,
                          int iLocal
);
```

The second argument should be true for local variables. The counting of local variables are reset whenever the program enters a new locality. Localities can not be nested.

Also note that local variables are allocated in a different segment because they are deallocated whenever the syntax analyzer leaves a locality.

1.8.15 _new_SymbolUF()

This function should be used to create a new user defined function symbol.

```
pSymbolUF _new_SymbolUF(peXobject pEx
);
```

A user function is defined by its serial number (serial number is actually not used in the current sytsem) and by the node number where the function actually starts.

The number of arguments and the number of local variables are defined in the generated command and not in the symbol table. This way these numbers are available as they should be during run time.

1.8.16 _new_eNODE()

This function should be used to create a new `eNODE`.

```
peNODE _new_eNODE(peXobject pEx
);
```

Each `eNODE` and `eNODE_l` structure has a serial number. The `eNODE`s are referencing each other using pointers. However after build these pointers become integer numbers that refer to the ordinal number of the node. Nodes are stored in a single memory block after they are packed during build.

An `eNODE` is a structure that stores a unit of compiled code. For example an addition in an expression is stored in an `eNODE` containing the code for the addition operator and containing pointers to the operands.

1.8.17 _new_eNODE_l()

This function should be used to create a new `eNODE` list. This is nothing else than a simple structure having two pointers. One pointer points to an `eNODE` while the other points to the next `eNODE_l` struct or to NULL if the current `eNODE_l` is the last of a list.

```
peNODE_l _new_eNODE_l(peXobject pEx,
                      char *pszFileName,
                      long lLineNumber
);
```

Note that `eNODE` and `eNODE_l` are converted to the same type of structure during build after the syntactical analysis is done.

1.8.18 ex_free()

This function releases all memory that was allocated during syntax analysis.

```
void ex_free(peXobject pEx
);
```

1.8.19 ex_init()

This function should be called before starting syntactical analysis. This function

positions the lexeme pointer to the first lexeme,

initializes the memory segments needed for structured memory allocation,

created the symbol tables

initializes 'class' variables

initializes the name space to be `main::`

```
int ex_init(peXobject pEx
);
```

1.8.20 ex_CleanNamePath()

This function created a normalized name space name from a non normalized. This is a simple string operation.

Think of name space as directories and variables as files. A simple variable name is in the current name space. If there is a 'path' before the variable or function name the path has to be used. This path can either be relative or absolute.

File system:

`../` is used to denote the parent directory in file systems.

Name space:

`_::` is used to denote the parent name space.

File system:

`mydir/../yourdir` is the same as `yourdir`

Name space:

`myns::_::yourns` is the same as `yourns`

This function removes the unneccesary downs and ups from the name space and creates the result in the same buffer as the original. This can always be done as the result is always shorter. (Well, not longer.)

```
void ex_CleanNamePath(char *s
);
```

1.8.21 ex_ConvertName()

Use this function to convert a relative name to absolute containing name space.

This function checks if the variable or function name is relative or absolute. If the name is relative it creates the absolute name using the current name space as a base.

The result is always put into the `Buffer`.

A name is relative if it does NOT contain `::` at all (implicit relative), if it starts with `::` or is it starts with `_::` (explicit relative).

```
int ex_ConvertName(char *s,          /* name to convert            */
                   char *Buffer,     /* buffer to store the result */
                   size_t cbBuffer,  /* size of the buffer         */
                   peXobject pEx     /* current expression object  */
);
```

The error value is `EX_ERROR_SUCCESS` (zero) menaing succesful conversion or `EX_ERROR_TOO_LONG_VARIABLE` meaning that the variable is too long for the buffer.

Note that the buffer is allocated in See ⟨undefined⟩ [ex_init()], page ⟨undefined⟩ according to the size value given in the class variable `cbBuffer`, which should be set by the main function calling syntax analysis.

1.8.22 ex_IsBFun()

This function checks if the current lexeme is a built-in function and returns pointer to the function in the table `BuiltInFunctions` or returns NULL if the symbol is not a built-in function.

```
pBFun ex_IsBFun(peXobject pEx
);
```

1.8.23 ex_IsUnop()

This function checks if the current lexeme is an unary operator and returns the op code or zero if the lexem is not an unary operator.

```
unsigned long ex_IsUnop(peXobject pEx
);
```

1.8.24 ex_IsBinop()

This function checks if the current lexeme is a binary operator of the given precedence and returns the op code or zero.

```
unsigned long ex_IsBinop(peXobject pEx,
                         unsigned long precedence
);
```

1.8.25 ex_LeftValueList()

This function works up a `leftvalue_list` pseudo terminal and creates the nodes for it.

```
peNODE_l ex_LeftValueList(peXobject pEx
);
```

1.8.26 ex_ExpressionList()

This function works up an `expression_list` pseudo terminal and creates the nodes for it.

```
peNODE_l ex_ExpressionList(peXobject pEx
);
```

1.8.27 ex_Local()

This function work up a `local` pseudo terminal. This does not create any node.

```
int ex_Local(peXobject pEx
);
```

The return value is `0` if no error happens.

`1` means sytax error (the coming token is not a symbol)

`2` means that there is no local environment (aka. the `local var` is not inside a function)

1.8.28 ex_LocalList()

This function work up a `local_list` pseudo terminal. This does not generate any node.

```
int ex_LocalList(peXobject pEx
);
```

The return value is `0` if no error happens.

`1` means sytax error (the coming token is not a symbol)

`2` means that there is no local environment (aka. the `local var` is not inside a function)

1.8.29 ex_Global()

This function work up a `global` pseudo terminal. This does not create any node.

```
int ex_Global(peXobject pEx
);
```

The return value is `0` if no error happens or the error is semantic and was reported (global variable redefinition).

`1` means syntax error (the coming token is not a symbol)

1.8.30 ex_GlobalList()

This function work up a `global_list` pseudo terminal. This does not generate any node.

```
int ex_GlobalList(peXobject pEx
);
```

The return value is `0` if no error happens.

`1` means sytax error (the coming token is not a symbol)

`2` means the variable was already defined

1.8.31 ex_LookupUserFunction()

This function searches a user defined function and returns a pointer to the symbol table entry. If the second argument `iInsert` is true the symbol is inserted into the table and an undefined function is created. This is the case when a function is used before declared. If the argument `iInsert` is fales `NULL` is returned if the function is not yet defined.

```
void **ex_LookupUserFunction(peXobject pEx,
                             int iInsert
);
```

1.8.32 ex_LookupGlobalVariable

This function searches the global variable symbol table to find the global variable with the name stored in TpEx->Buffer. If the variable was not declared then this function inserts the variable into the symbol table if the argument `iInsert` is true, but nothing more: the symbol table entry remains `NULL`.

```
void **ex_LookupGlobalVariable(peXobject pEx,
                               int iInsert
);
```

The function returns pointer to the pointer stored in the symbol table associated with the global variable.

1.8.33 ex_LookupLocallyDeclaredGlobalVariable

This function searches the global variable symbol table to find the global variable with the name stored in TpEx->Buffer. If the variable was not declared then this function return `NULL`. Othervise it returns a pointer to a `void *` pointer, which is `NULL`.

Note that this table is allocated when the program starts a `sub` or `function` (aka. when we go local) and is used to register, which variables did the program declare as global variables inside the subroutine. There is no any value associated with the symbols in this table, as the symbols are also inserted into the global symbol table which serves the purpose.

```
void **ex_LookupLocallyDeclaredGlobalVariable(peXobject pEx
);
```

The function returns pointer to the pointer stored in the symbol table associated with the global variable or `NULL`.

1.8.34 ex_LookupLocalVariable

This function searches the local variable symbol table to find the local variable with the name stored in TpEx->Buffer. If the variable was not declared and the argument `iInsert` is true then then this function inserts the variable into the symbol table, but nothing more: the symbol table entry remains `NULL`.

```
void **ex_LookupLocalVariable(peXobject pEx,
                              int iInsert
);
```

The function returns pointer to the pointer stored in the symbol table associated with the global variable.

1.8.35 ex_Tag

This function implements the syntax analysis for the lowest syntax elements of an expression. This function is called when syntax analysis believes that a TAG has to be worked up in an expression. A tag is defined formally as

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

peNODE ex_Tag(peXobject pEx
);
```

The function returns pointer to the new node.

1.8.36 ex_Expression_i

This function is called to analyze a sub-expression that has no lower precedence operators than `i` (unless enclosed in parentheses inside the sub expression).

If the argument variable `i` is `1` then this function simply calls See ⟨undefined⟩ [ex_Tag], page ⟨undefined⟩. Otherwise it calls itself recursively twice with optionally compiling the operator between the two subexpressions.

```
peNODE ex_Expression_i(peXobject pEx,
                       int i
);
```

The function returns pointer to the new node.

1.8.37 ex_Expression_r

This function implements the syntax analysis for an expression. This is quite simple. It only calls See ⟨undefined⟩ [ex_Expression_i], page ⟨undefined⟩ to handle the lower precendece expression.

```
void ex_Expression_r(peXobject pEx,
                     peNODE *Result
);
```

1.8.38 ex_IsSymbolValidLval(pEx)

This function checks whether the actual symbol used in as a start symbol of a left value is defined as a CONST in the BASIC program or not. If this is a const then the syntax analizer has to report an error (since v1.0b31).

This function is called from the function See ⟨undefined⟩ [ex_LeftValue], page ⟨undefined⟩ after the symbol was name space corrected.

Note that a symbol can be a global, name space independant constant, a name space local constant and a function local constant. All these differ only in name decoration inside the interpreter.

If a symbol is a local variable but is also a module or global symbol, but is NOT a function local symbol then that variable can indeed stand on the left side of a LET command. Therefore we check if the symbol is in the local variables table and in case this is in some of the global or module contant table, we just do not care.

```
int ex_IsSymbolValidLval(peXobject pEx
);
```

The function returns 1 if the symbol is a constant or zero if not.

1.8.39 ex_LeftValue

This function implements the syntax analisys for a left value.

```
peNODE ex_LeftValue(peXobject pEx
);
```

The function returns pointer to the new node.

1.8.40 ex_PredeclareGlobalLongConst()

This function is used to declare the global constants that are given in the syntax defintinon, and should be defined before the program is started to be analized.

```
int ex_PredeclareGlobalLongConst(peXobject pEx,
                                 char *pszConstName,
                                 long lConstValue
);
```

1.8.41 ex_IsCommandThis

This is the most general syntax analysis function that tries to match the syntax of the actual line syntax provided in argument `p` against the token list at the actual position.

The function has several side effects altering optionally the global and local variable table, define user defined functions and so on.

The function signals the success of its operation via the argument `piFailure` setting the `int` pointed by it to be zero or the error code.

If the syntax does not match the token list then the function cleans up all its actions if possible to allow the caller to iterate over to the next syntax defintion. In such a situation `*piFailure` is set `EX_ERROR_SYNTAX`

If the syntax does not match the token list but the analysis went too far and had side effects that cannot be reversed then no cleanup is made. In such a situation `*piFailure` is set `EX_ERROR_SYNTAX_FATAL`.

`*piFailure` is also set to this value if the syntax definition reaches a "star" point. If the syntax analysis matches a line up to a "star" point then the line should match that syntax definition or is known erroneous. For example a command starting with the two keywords `'declare' 'command'` after these two keywords reach a "star" point because no other line syntax but extrenal command declaration starts with these two keywords. In such a situation signalling fatal syntax error saves the compiler time to check other syntax definition.

A "star" point is named this way, because the file `syntax.def` uses the character `*` to denote this point in the syntax definitions.

```
peNODE ex_IsCommandThis(peXobject pEx,
                        pLineSyntax p,
                        int *piFailure
);
```

If the syntax analysis fully matches the syntax definition provided in the argument then the function returns the node that was generated. If more then one nodes were generated during the syntax analysis of the line then the root node of the generated nodes is returned.

1.8.42 ex_Command_r()

This function finds the matching sytax line for the actual line in a loop. It starts with the first syntax definition and goes on until there are no more syntax defintions, a fatal error has happened or the actual line is matched.

```
void ex_Command_r(peXobject pEx,
                  peNODE *Result,
                  int *piFailure
);
```

`pEx` is the execution object.

`Result` is the resulting node.

`piFailure` is the error code.

1.8.43 ex_Command_l()

This function goes over the source lines and performs the syntax analysis. This function calls the function See ⟨undefined⟩ [ex_Command_r()], page ⟨undefined⟩. When that function returns it allocated the list nodes that chain up the individual lines. It also defines the labels that are waiting to be defined.

```
int ex_Command_l(peXobject pEx,
                 peNODE_l *Result
);
```

When all the lines are done this function cleans the name space stack, check for undefined labels that remained undefined still the end of the source file.

1.8.44 ex_Pragma

This function implements the compiler directive "declare option".

When the compiler finds a "declare option" directive it calls this function. The first argument is the compiler class pointer. The second argument points to a constant string containing the option.

The function implements the internal settings of the compiler options reflecting the programmer needs expressed by the option. For example DeclareVars will require all variables declared to be either global or local.

If the programmer specified an option, which is not implemented the error reporting function is called.

```
int ex_Pragma(peXobject pEx,
              char *pszPragma
);
```

The function returns `0` when the option was processed, and `1` when not implemented option was supplied as argument.

1.8.45 ex_IsCommandCALL()

Because the syntax of a call statement is very special here is a special function to analyze the CALL statement.

A call statement is a keyword CALL followed by a function call.

If the function or sub is already defined then the keyword CALL can be missing.

When the function or sub is called this way and not inseide an expression the enclosing parentheses can be missing.

```
peNODE ex_IsCommandCALL(peXobject pEx,
                        pLineSyntax p,
                        int *piFailure
);
```

To get some description of waiting labels see the description of the function See ⟨undefined⟩ [ex_PushWaitingLabel()], page ⟨undefined⟩.

1.8.46 ex_IsCommandOPEN()

The open statement is a simple one. The only problem is that the last parameter defining the length of a record is optional. This can only be handled using a separate function

```
peNODE ex_IsCommandOPEN(peXobject pEx,
                        pLineSyntax p,
                        int *piFailure
);
```

'open' expression 'for' absolute_symbol 'as' expression 'len' '=' expression nl

1.8.47 ex_IsCommandSLIF()

If syntax analysis gets to calling this function the command is surely not single line if, because the command SLIF is recognised by `IsCommandIF`.

The syntax of the command IF is presented in the syntax table before SLIF and therefore if the syntax analyser gets here it can not be SLIF.

The original function `IsCommandThis` could also do failing automatically, but it is simpler just to fail after the function call, so this function is just a bit of speedup.

```
peNODE ex_IsCommandSLIF(peXobject pEx,
                        pLineSyntax p,
                        int *piFailure
);
```

1.8.48 ex_IsCommandIF()

The statement IF is quite simple. However there is another command that has almost the same syntax as the IF statement. This is the SLIF, single line IF.

The difference between the command IF and SLIF is that SLIF does not have the new line character after the keyword `THEN`.

```
peNODE ex_IsCommandIF(peXobject pEx,
                      pLineSyntax p,
                      int *piFailure
);
```

IF/IF: 'if' * expression 'then' go_forward(IF) nl SLIF/SLIF: 'slif' * expression 'then'

1.8.49 ex_IsCommandLET()

```
peNODE ex_IsCommandLET(peXobject pEx,
                       pLineSyntax p,
                       int *piFailure
);
```

1.9 builder.c

This module can and should be used to create the memory image for the executor module from the memory structure that was created by the module `expression`.

The memory structure created by `expression` is segmented, allocated in many separate memory chunks. When the module `expression` has been finished the size of the memory is known. This builder creates a single memory chunk containing all the program code.

Note that the function names all start with the prefix `build_` in this module.

The first argument to each function is a pointer to a `BuildObject` structure that contains the "global" variables for the module. This technique is used to ensure multithread usage. There are no global variables which are really global within the process.

The functions in this module are:

=toc

1.9.1 The structure of the string table

The string table contains all string contansts that are used in the program. This includes the single and multi line strings as well as symbols. (note that even the variable name after the keyword `next` is ignored but stored in the string table).

The strings in the string table are stored one after the other zero character terminated. Older version of ScriptBasic v1.0b21 and before stored string constants zero character terminated. Because of this string constants containing zero character were truncated (note that `\000` creates a zero character in a string constant in ScriptBasic).

The version v1.0b22 changed the way string constants are stored and the way string table contains the strings. Each string is stored with its length. The length is stored as a `long` on `sizeof(long)` bytes. This is followed by the string. Whenever the code refers to a string the byte offset of the first character of the string is stored in the built code. For example the very first string starts on the 4. byte on 32 bit machines.

Altough the string length and zero terminating characters are redundant information both are stored to avoid higher level mistakes causing problem.

1.9.2 build_AllocateStringTable()

This function allocates space for the string table. The size of the string table is already determined during syntax analysis. The determined size should be enough. In some cases when there are repeated string constants the calculated sizte is bigger than the real one. In that case the larger memory is allocated and used, but only the really used part is written to the cache file.

If the program does not use any string constants then a dummy string table of length one byte is allocated.

```
void build_AllocateStringTable(pBuildObject pBuild,
                               int *piFailure
);
```

The first argument is the usual pointer to the "class" structure. The second argument is the result value. It can have two values:

`BU_ERROR_SUCCESS` which is guaranteed zero, means the function was successful.

`BU_ERROR_MEMORY_LOW` means the memory allocation function could not allocate the neccessary memory

The string table is allocated using the function `alloc_Alloc`. The string table is pointed by the class variable `StringTable`. The size of the table is stored in `cStringTable`

1.9.3 build_StringIndex()

In the built code all the strings are references using the offset of the string from the string table (See See ⟨undefined⟩ [build_AllocateStringTable()], page ⟨undefined⟩). This function calculates this value for the string.

This function is used repetitively during the code building. Whenever a string index is sought that is not in the string table yet the string is put into the table and the index is returned.

If there is not enough space in the string table the function calls the system function `exit` and stops the process. This is rude especially in a multithread application but it should not ever happen. If this happens then it is a serious internal error.

```
unsigned long build_StringIndex(pBuildObject pBuild,
                                char *s,
                                long sLen
);
```

1.9.4 build_Build_l()

This function converts an `eNODE_l` list to `cNODE` list in a loop. This function is called from See ⟨undefined⟩ [build_Build()], page ⟨undefined⟩ and from See ⟨undefined⟩ [build_Build_r()], page ⟨undefined⟩.

```
int build_Build_l(pBuildObject pBuild,
                  peNODE_l Result
);
```

The function returns the error code, or zero in case of success.

1.9.5 build_Build_r()

This function builds a single node. This actually means copiing the values from the data structure created by the module `expression`. The major difference is that the pointers of the original structure are converted to `unsigned long`. Whenever a pointer pointed to a `eNODE` the `unsigned long` will contain the `NodeId` of the node. This ID is the same for the `eNODE` and for the `cNODE` that is built from the `eNODE`.

```
int build_Build_r(pBuildObject pBuild,
                  peNODE Result
);
```

The node to be converted is passed by the pointer `Result`. The return value is the error code. It is zero (`BU_ERRROR_SUCCESS`) in case of success.

When the node pointed by `Result` references other nodes the function recursively calls itself to convert the referenced nodes.

1.9.6 build_Build()

This is the main entry function for this module. This function initializes the class variable pointed by `pBuild` and calls See ⟨undefined⟩ [build_Build_1()], page ⟨undefined⟩ to build up the command list.

```
int build_Build(pBuildObject pBuild
);
```

1.9.7 build_MagicCode()

This is a simple and magical calculation that converts any ascii date to a single unsigned long. This is used as a magic value in the binary format of the compiled basic code to help distinguish incompatible versions.

This function also fills in the sVersion static struct that contains the version info.

```
unsigned long build_MagicCode(pVersionInfo p
);
```

1.9.8 build_SaveCCode()

This function saves the binary code of the program into the file given by the name `szFileName` in C programming language format.

The saved file can be compiled using a C compiler on the platform it was saved. The generated C file is not portable.

```
void build_SaveCCode(pBuildObject pBuild,
                     char *szFileName
);
```

1.9.9 build_SaveCorePart()

This function saves the binary content of the compiled file into an already opened file. This is called from both `build_SaveCode` and from `build_SaveECode`.

Arguments:

`pBuild` is the build object

`fp` is the `FILE *` file pointer to an already binary write opened (`"wb"`) file.

The file `fp` is not closed even if error occures while writing the file.

```
int build_SaveCorePart(pBuildObject pBuild,
                       FILE *fp,
                       unsigned long fFlag
```

```
);
```

The function returns `BU_ERROR_SUCCESS` (zero) if there was no error or `BU_ERROR_FAIL` if the function fails writing the file.

1.9.10 build_SaveCore()

This function saves the binary content of the compiled file into an already opened file. This is called from both `build_SaveCode` and from `build_SaveECode`.

Arguments:

`pBuild` is the build object

`fp` is the `FILE *` file pointer to an already binary write opened (`"wb"`) file.

The file `fp` is not closed even if error occures while writing the file.

```
int build_SaveCore(pBuildObject pBuild,
                   FILE *fp
);
```

The function returns `BU_ERROR_SUCCESS` (zero) if there was no error or `BU_ERROR_FAIL` if the function fails writing the file.

1.9.11 build_SaveCode()

This function saves the binary code of the program into the file given by the name `szFileName`.

This version is hard wired saving the code into an operating system file because it uses `fopen`, `fclose` and `fwrite`. Later versions may use other compatible functions passed as argument and thus allowing output redirection to other storage media (a database for example).

However I think that this code is quite simple and therefore it is easier to rewrite the whole function along with See ⟨undefined⟩ [build_LoadCode()], page ⟨undefined⟩ for other storage media than writing an interface function.

The saved binary code is NOT portable. It saves the internal values as memory image to the disk. It means that the size of the code depends on the actual size of long, char, int and other types. The byte ordering is also system dependant.

The saved binary code can only be loaded with the same version, and build of the program, therefore it is vital to distinguish each compilation of the program. To help the recognition of the different versions, the code starts with a version structure.

The very first byte of the code contains the size of the long on the target machine. If this is not correct then the code was created on a different processor and the code is incompatible.

The version info structure has the following fileds:

`MagicCode` is a magic constant. This contains the characters BAS and a character 1A that stops output to screen on DOS operating systems.

`VersionHigh` The high part of the version of the STANDARD version.

`VersionLow` The low part of the version of the STANDARD version.

`MyVersionHigh` The high part of the version of the variation. This is always zero for the STANDARD version.

`MyVersionLow` The low part of the version of the variation. This is always zero for the STANDARD version.

`Build` A build code which is automatically calculated from the compilation date.

`Variation` 8 characters (NOT ZERO TERMINATED!) naming the version "STANDARD" for the STANDARD version (obvious?)

```
int build_SaveCode(pBuildObject pBuild,
                   char *szFileName
);
```

The function returns zero on success (`BU_ERROR_SUCCESS`) and `BU_ERROR_FAIL` if the code could not be saved.

1.9.12 build_SaveECode()

This function saves the binary code of the program into the file given by the name `szFileName` in exe format.

This is actually nothing but the copy of the original interpreter file and the binary code of the BASIC program appended to it and some extra information at the end of the file to help the reader to find the start of the binary BASIC program when it tries to read the exe file.

```
void build_SaveECode(pBuildObject pBuild,
                     char *pszInterpreter,
                     char *szFileName
);
```

1.9.13 build_GetExeCodeOffset()

This function checks that the actually running exe contains the binary BASIC program attached to its end. It returns zero if not, otherwise it returns 1.

The argument `pszInterpreter` should be `argv[0]` thus the code can open the executable file and check if it really contains the BASIC code

`plOffset` should point to a long variable ready to recieve the file offset where the BASIC code starts

`plEOFfset` should point to a long variable ready to receive the file offset where the BASIC code finishes. This is the position of the last byte belonging to the BASIC code, thus if `ftell(fp) > *plEOFfset` means the file pointer is after the code and should treat it as EOF condition when reading the BASIC program code.

It is guaranteed that both `*plOffset` and `*plEOFfset` will be set to `0` (zero) if the file proves to be a standard BASIC interpreter without appended BASIC code.

```
int build_GetExeCodeOffset(char *pszInterpreter,
                           long *plOffset,
                           long *plEOFfset
);
```

1.9.14 build_LoadCore()

This function loads the binary code from an opened file.

Arguments:

`pBuild` is the build object

`szFileName` is the name of the file that is opened. Needed for reporting purposes.

`fp` opened `FILE *` file pointer opened for binary reading (aka `"rb"`), and positioned where the BASIC code starts.

`lEOFfset` should be the position of the last byte that belongs to the BASIC code so that `ftell(fp)>lEOFfset` is treated as EOF condition. If this value is zero that means that the BASIC code is contained in the file until the physical end of file.

```
void build_LoadCore(pBuildObject pBuild,
                    char *szFileName,
                    FILE *fp,
                    long lEOFfset
);
```

Note that the program does not return error code, but calls the reporting function to report error. The file `fp` is not closed in the function even if error has happened during reading.

1.9.15 build_LoadCodeWithOffset()

For detailed definition of the binary format see the code and the documentation of See ⟨undefined⟩ [build_SaveCode()], page ⟨undefined⟩

In case the file is corrupt the function reports error.

```
void build_LoadCodeWithOffset(pBuildObject pBuild,
                              char *szFileName,
                              long lOffset,
                              long lEOFfset
);
```

1.9.16 build_LoadCode()

For detailed definition of the binary format see the code and the documentation of See ⟨undefined⟩ [build_SaveCode()], page ⟨undefined⟩

In case the file is corrupt the function reports error.

```
void build_LoadCode(pBuildObject pBuild,
                    char *szFileName
);
```

1.9.17 build_IsFileBinaryFormat()

This function test a file reading its first few characters and decides if the file is binary format of a basic program or not.

```
int build_IsFileBinaryFormat(char *szFileName
);
```

1.9.18 build_pprint()

This is a debug function that prints the build code into a file.

This function is not finished and the major part of it is commented out using `#if 0` construct.

```
void build_pprint(pBuildObject pBuild,
                  FILE *f
);
```

1.9.19 build_CreateFTable()

When the binary code of the BASIC program is saved to disk the symbol table of the user defined functions and the symbol table of global variables is also saved. This may be needed by some applications that embed ScriptBasic and want to call specific function or alter global variables of a given name from the embedding C code. To do this they need the serial number of the global variable or the entry point of the function. Therefore ScriptBasic v1.0b20 and later can save these two tables into the binary code.

The format of the tables is simple optimized for space and for simplicity of generation. They are stored first in a memory chunk and then written to disk just as a series of bytes.

The format is

```
long      serial number of variable or entry point of the function
zchar     zero character terminated symbol
```

This is easy to save and to load. Searching for it is a bit slow. Embedding applications usually have to search for the values only once, store the serial number/entry point value in their local variable and use the value.

The function `CreateFTable` converts the symbol table of user defined function collected by symbolic analysis into a single memory chunk.

The same way See ⟨undefined⟩ [build_CreateVTable()], page ⟨undefined⟩ converts the symbol table of global variables collected by symbolic analysis into a single memory chunk.

```
int build_CreateFTable(pBuildObject pBuild
);
```

1.9.20 build_CreateVTable()

When the binary code of the BASIC program is saved to disk the symbol table of the user defined functions and the symbol table of global variables is also saved. This may be needed by some applications that embed ScriptBasic and want to call specific function or alter global variables of a given name from the embedding C code. To do this they need the serial number of the global variable or the entry point of the function. Therefore ScriptBasic v1.0b20 and later can save these two tables into the binary code.

The format of the tables is simple optimized for space and for simplicity of generation. They are stored first in a memory chunk and then written to disk just as a series of bytes.

The format is

```
long      serial number of variable or entry point of the function
zchar     zero character terminated symbol
```

This is easy to save and to load. Searching for it is a bit slow. Embedding applications usually have to search for the values only once, store it in their local variable and use the value.

The function See ⟨undefined⟩ [build_CreateFTable()], page ⟨undefined⟩ converts the symbol table of user defined function collected by symbolic analysis into a single memory chunk.

The same way `CreateVTable` converts the symbol table of global variables collected by symbolic analysis into a single memory chunk.

```
int build_CreateVTable(pBuildObject pBuild
);
```

1.9.21 build_LookupFunctionByName()

```
long build_LookupFunctionByName(pBuildObject pBuild,
                           char *s
);
```

1.9.22 build_LookupVariableByName()

```
long build_LookupVariableByName(pBuildObject pBuild,
                           char *s
);
```

1.10 reader.c

This module contains the functions that read a source file.

Script basic has several passes until it can start to execute the code. The very first pass is to read the source lines from the files. The routines in this module perform this task and build up a linked list that contains the ascii values of the lines.

The input functions are parametrized, and the caller should support. If you have different system dependent file reading functions, or if you have the input file in some format in memory or in any other data holding space you can support these routines with character fetch functions.

1.10.1 reader_IncreaseBuffer()

When the reader encounters a line which is longer than the currently allocated input buffer it calls this function to increase the size of the input buffer. The input buffer is linearly increased by `BUFFER_INCREMENT` size (defined in the header section of `reader.c`

When a new buffer is allocated the bytes from the old buffer are copied to the new and the old buffer is released. It is vital that the buffer is always referenced via the pRo->buffer pointer because resizing buffer does change the location of the buffer.

If the memory allocation fails the function return `READER_ERROR_MEMORY_LOW` error. Otherwise it returns zero.

```
int reader_IncreaseBuffer(pReadObject pRo
);
```

1.10.2 reader_gets()

This function reads a newline terminated line from the file. The file is identified by function pRo->fpGetCharacter and the pointer `fp`.

When the input buffer is too small it automatically increases the buffer. The terminating new line is included in the buffer. If the last line of the file is not terminated by newline an extra newline character is added to this last line.

The addition of this extra newline character can be switched off setting pRo->fForceFinalNL to false. Even if this variable is false the normal newline characters which are present in the file are included in the buffer.

```
int reader_gets(pReadObject pRo,
                void *fp
);
```

1.10.3 reader_ReadLines()

This function calls See ⟨undefined⟩ [reader_ReadLines_r()], page ⟨undefined⟩ to read the lines of the file given by the file name `szFileName` into pRo->Result. For further information see See ⟨undefined⟩ [reader_ReadLines_r()], page ⟨undefined⟩.

```
int reader_ReadLines(pReadObject pRo,
                     char *szFileName
);
```

The function returns zero or the error code.

1.10.4 reader_ReadLines_r()

This function reads the lines of a file and creates a linked list of the read lines.

```
int reader_ReadLines_r(pReadObject pRo,
                       char *szFileName,
                       pSourceLine *pLine
);
```

The file is identified by its name given in the string variable `szFileName`. The file is opened by the function pointed by pRo->fpOpenFile This function should return a void pointer and this void pointer is passed to See ⟨undefined⟩ [reader_gets()], page ⟨undefined⟩ (`reader_gets`) to get a single character.

The argument `pLine` is a pointer to a `SourceLine` pointer. The linked list lines read will be chained into this pointer. The last read line will be followed by the line pointed by `*pLine` and `*pLine` will point to the first line.

This design makes it easy to use and elegant to perform file inclusions. The caller has to pass the address of the pointer field `next` of the source line after which the file is to be inserted.

See also `ReadLines` that calls this function.

1.10.5 reader_ProcessIncludeFiles()

This function is called from See ⟨undefined⟩ [reader_ReadLines()], page ⟨undefined⟩ after calling See ⟨undefined⟩ [reader_ReadLines_r()], page ⟨undefined⟩.

This function goes through all the lines and checks if there is any line containing an include directive.

An include directive is a line starting with a word INCLUDE (case insensitive) and is followed by the file name. The file name can be enclodes between double quotes.

Note that the processing of the include directives are done on the characters on the line, because they are processed before any tokenization of the lexer module. This can cause some problem only when there is an include like line inside a multiline string. For example:

```
a = """Hey this is a multiline string
include "subfile.txt"
"""
```

This **will** include the file `subfile.txt` and its content will become part of the string. This becomes more complicated when the file `subfile.txt` contains strings.

The file name may not be enclosed between double quotes. In this case the file is tried to be found in predefined system directories.

If the programmer uses the command IMPORT instead of INCLUDE the file will only be included if it was not included yet into the current program.

```
void reader_ProcessIncludeFiles(pReadObject pRo,
                                pSourceLine *pLine
);
```

The file read is inserted into the plce where the include statement was.

1.10.6 reader_LoadPreprocessors()

Preprocessors are not part of ScriptBasic. They can be implemented as external DLLs and should be configured in the configuration file.

When a line contains

```
USE preprocessorname
```

this reader module loads the preprocessor DLL or SO (dll under unix) file.

```
void reader_LoadPreprocessors(pReadObject pRo,
                                pSourceLine *pLine
);
```

1.10.7 reader_StartIteration()

The package supports functions that help upper layer modules to iterate through the lines read. This function should be called to start the iteration and to set the internal iteration pointer to the first line.

```
void reader_StartIteration(pReadObject pRo
);
```

1.10.8 reader_NextLine()

This function returns a string which is the next line during iteration. This function does NOT read anything from any file, only returns a pointer to a string that was already read.

This function can be used together with See ⟨undefined⟩ [reader_NextCharacter()], page ⟨undefined⟩. When a line was partially passed to an upper layer that uses `reader_NextCharacter` this function will only return the rest of the line.

```
char *reader_NextLine(pReadObject pRo
);
```

1.10.9 reader_NextCharacter()

This function gets the next character from the actual line, or gets the first character of the next line.

This function does NOT read anything from any file, only returns a character from a string that was already read.

When the last character of the last line was passed it return `EOF`

```
int reader_NextCharacter(void *p
);
```

1.10.10 reader_FileName()

This function returns the file name of the actual line. This is the string that was used to name the file when it was opened. This can be different for different lines when the reader is called several times to resolve the "include" statements.

```
char *reader_FileName(void *p
);
```

1.10.11 reader_LineNumber()

This function returns the line number of the current line durig iteration. This number identifies the line in the file where it was read from.

```
long reader_LineNumber(void *p
);
```

1.10.12 reader_InitStructure()

This function should be called to initialize the reader structure. It sets the file handling routines to the standard `fopen`, `fclose` and `getc` functions, and also sets the function pointers so that the module uses `malloc` and `free`.

```
void reader_InitStructure(pReadObject pRo
);
```

1.10.13 reader_RelateFile()

This function gets a file name, which is either absolute or relative to the current working directory and another file name which is absolute or relative to the first one.

The return value of the function is a file name which is either absolute or relative to the current woring directory.

The return value is dynamically allocated and is to be release by the caller. The allocation function is taken from the class function and the segment is `pMemorySegment`.

```
char *reader_RelateFile(pReadObject pRo,
                        char *pszBaseFile,
                        char *pszRelativeFile
);
```

1.10.14 reader_DumpLines()

This is a debug function that prints the lines into a debug file.

```
void reader_DumpLines(pReadObject pRo,
                      FILE *fp
);
```

1.11 myalloc.c

1.11.1 Multi-thread use of this module

You can use this module in multi threaded environment. In this case the module depend on the module `thread.c` which contains the thread and mutex interface functions that call the operating system thread and mutex functions on UNIX and on Windows NT.

In single thread environment there is no need to use the locking mechanism. To get a single-thread version either you can edit this file (`myalloc.c`) or compile is using the option `-DMTHREAD=0` The default compilation is multi threaded.

Multi thread implementation has two levels. One is that the subroutines implemented in this module call the appropriate locking functions to ensure that no two concurrent threads access and modify the same data at a time and thus assure that the data of the module is correct. The other level is that you can tell the module that the underlying memory allocation and deallocation modules are mot thread safe. There are global variables

implementing global mutexes that are locked and unlocked if you use the module that way. This can be useful in some environment where `malloc` and `free` are not thread safe.

Note that this should not be the case if you call `malloc` and `free` or you linked the wrong versio of libc. However you may use a non-thread safe debug layer for example the one that ScriptBasic uses.

1.11.2 alloc_InitSegment()

Call this function to get a new segment. You should specify the functions that the segement should use to get memory from the operating system, and the function the segment should use to release the memory to the operating system. These functions should be like `malloc` and `free`.

If the second argument is `NULL` then the function will treat the first argument as an already allocated and initialized memory segment and the memory allocation and freeing functions will be inherited from that segment.

```
void *alloc_InitSegment(void * (*maf)(size_t), /* a 'malloc' and a 'free' like func
                        void   (*mrf)(void *)
);
```

The return value is a `void*` pointer which identifies the segment and should be passed to the other functions as segment argument.

The first argument is the `malloc` like function and the second if the `free` like function.

1.11.3 alloc_GlobalUseGlobalMutex()

Some installation use memory allocation function that are not thread safe. On some UNIX installations `malloc` is not thread safe. To tell the module that all the allocation function primitives are not thread safe call this function before initializing any segment.

```
void alloc_GlobalUseGlobalMutex(
);
```

1.11.4 alloc_SegmentLimit()

You can call this function to set a segment limit. Each segment keeps track of the actual memory allocated to the segment. When a new piece of memory allocated in a segment the calculated segment size is increased by the size of the memory chunk. When a piece of memory is release the calculated size of the segment is decreased.

Whenever a segment approaches its limit the next allocation function requesting memory that would exceed the limit returns `NULL` and does not allocate memory.

The value of the limit is the number of bytes allowed for the segment. This is the requested number of bytes without the segment management overhead.

Setting the limit to zero means no limit except the limits of the underlying memory allocation layers, usually `malloc`.

You can dynamically set the limit during handling the memory at any time except that you should not set the limit to zero unless the segment is empty and you should not set

the limit to a positive value when the actual limit is zero (no limit) and the segment is not empty. This restriction is artificial in this release but is needed to be followed to be compatible with planned future developments.

This function sets the limit for the segment pointed by `p` and returns the old value of the segment.

```
long alloc_SegmentLimit(void *p,
                        unsigned long NewMaxSize
);
```

1.11.5 alloc_FreeSegment()

Use this function to release all the memory that was allocated to the segment `p`. Note that after calling this function the segment is still usable, only the memory that it handled was released. If you do not need the segment anymore call the function See ⟨undefined⟩ [alloc_FinishSegment()], page ⟨undefined⟩ that calls this function and then releases the memory allocated to store the segment information.

Sloppy programmers may pass `NULL` as argument, it just returns.

```
void alloc_FreeSegment(void *p
);
```

1.11.6 alloc_FinishSegment()

Use this function to release all the memory that was allocated to the segment `p`. This function also releases the memory of the segment head and therefore the segment pointed by `p` is not usable anymore.

```
void alloc_FinishSegment(void *p
);
```

1.11.7 alloc_Alloc()

Use this function to allocate a memory piece from a segment.

```
void *alloc_Alloc(size_t n,
                  void *p
);
```

The first argument is the size to be allocated. The second argument is the segment which should be used for the allocation.

If the memory allocation fails the function returns `NULL`.

1.11.8 alloc_Free()

You should call this function whenever you want to release a single piece of memory allocated from a segment. Note that you also have to pass the segment pointer as the second argument, because the segment head pointed by this `void` pointer contains the memory releasing function pointer.

Sloppy programmers may try to release `NULL` pointer without harm.

```
void alloc_Free(void *pMem, void *p
);
```

1.11.9 alloc_Merge()

Call this function in case you want to merge a segment into another. This can be the case when your program builds up a memory structure in several steps.

This function merges the segment `p2` into `p1`. This means that the segment `p1` will contain all the memory pieces that belonged to `p2` before and `p2` will not contain any allocated memory. However the segment `p2` is still valid and can be used to allocated memory from. If you also want to finish the segment `p2` call the function See ⟨undefined⟩ [alloc_MergeAndFinish()], page ⟨undefined⟩.

```
void alloc_Merge(void *p1, void *p2
);
```

Note that the two segments SHOULD use the same, or at least compatible system memory handling functions! You better use the same functions for both segments.

Example:

ScriptBasic builds up a sophisticated memory structure during syntactical analysis. This memory structure contains the internal code generated from the program lines of the basic program. When ScriptBasic analyses a line it tries several syntax descriptions. It checks each syntax defintion against the tokens of the line until it finds one that fits. These checks need to build up memory structure. However if the check fails and ScriptBasic should go for the next syntac definition line to check the memory allocated during the failed checking should be released. Therefore these memory pieces are allocated from a segment that the program calls `pMyMemorySegment`. If the syntax check fails this segment if freed. If the syntax check succedes this segment is merged into another segement that contains the memory structures allocated from the previous basic program lines.

1.11.10 alloc_MergeAndFinish()

Use this function in case you not only want to merge a segment into another but you also want to finish the segment that was merged into the other.

See also See ⟨undefined⟩ [alloc_Merge()], page ⟨undefined⟩

```
void alloc_MergeAndFinish(void *p1, void *p2
);
```

1.11.11 alloc_InitStat()

This function initializes the global statistical variables. These variables can be used in a program to measure the memory usage.

This function should be called before any other memory handling function.

```
void alloc_InitStat(
);
```

1.11.12 alloc_GlobalGetStat()

From period to period the code using this memory management layer may need to know how much memory the program is using.

Calling this function from time to time you can get the minimum and maximum memory that the program used via this layer since the last call to this function or since program start in case of the first call.

```
void alloc_GlobalGetStat(unsigned long *pNetMax,
                         unsigned long *pNetMin,
                         unsigned long *pBruMax,
                         unsigned long *pBruMin,
                         unsigned long *pNetSize,
                         unsigned long *pBruSize
);
```

1.11.13 alloc_GetStat()

From period to period the code using this memory management layer may need to know how much memory the program is using.

Calling this function from time to time you can get the minimum and maximum memory that the program used via this layer since the last call to this function or since program start in case of the first call.

```
void alloc_GetStat(void *p,
                   unsigned long *pMax,
                   unsigned long *pMin,
                   unsigned long *pActSize
);
```

1.12 match.c

=abstract A simple, non-regular expression pattern matching module mainly to perform file name pattern matching, like `*.txt` or `file0?.bin` and alikes. =end

This is a simple and fast pattern matching algorithm. This can be used when the matching does not require regular expression complexity and the processign on the other hand should be fast.

There are two major tasks implemented here. One is to match a string against a pattern. The second is to create a replacement string. When a pattern is matched by a string an array of string values are created. Each contains a substring that matches a joker character. Combining this array and a format string a replacement string can be created.

For example:

```
String = "mortal  combat"
Pattern = "mo?tal co*"
```

the joker characters are the ?, the space (matching one or more space) and the * character. They are matched by `r`, two spaces and `mbat`. If we use the format string

```
Format string = "$1u$2"
```

we get the result string `rumbat`. The format string can contain `$n` placeholders where `n` starts with 1 and is replaced by the actual value of the n-th joker character.

1.12.1 match_index

There are a few characters that can be used as joker character. These are

```
*#$?&%!+/|<>
```

`match_index` returns the serial number of the character.

```
unsigned long match_index(char ch
);
```

1.12.2 InitSets

Call this function to initialize a set collection. The argument should point to a `MatchSets` structure and the function fills in the default values.

```
void match_InitSets(pMatchSets pMS
);
```

1.12.3 ModifySet

This function can be used to modify a joker set. The first argument `pMS` points to the joker set collection. The second argument `JokerCharacter` specifies the joker character for which the set has to be modified.

The argument `nChars` and `pch` give the characters that are to be modified in the set. `nChars` is the number of characters in the character array pointed by `pch`.

The last argument `fAction` specifies what to do. The following constants can be used in logical OR.

```
TO_HEADER:

#define MATCH_ADDC 0x0001 //add characters to the set
#define MATCH_REMC 0x0002 //remove characters from the set
#define MATCH_INVC 0x0004 //invert the character
#define MATCH_SNOJ 0x0008 //set becomes no-joker
#define MATCH_SSIJ 0x0010 //set becomes single joker
#define MATCH_SMUJ 0x0020 //set becomes multiple joker
#define MATCH_NULS 0x0040 //nullify the set
#define MATCH_FULS 0x0080 //fullify the set

*/
```

The function first checks if it has to modify the state of the joker character. If any of the bits `MATCH_SNOJ`, `MATCH_SSIJ` or `MATCH_SMUJ` is set in the field `fAction` the type of the set is modified.

If more than one bit of these is set then result is undefined. Current implementation checks these bits in a specific order, but later versions may change.

If the bit `MATCH_NULS` is set all the characters are removed from the set. If the bit `MATCH_FULS` is set all characters are put into the set.

If more than one bit of these is set then result is undefined. Current implementation checks these bits in a specific order, but later versions may change.

`MATCH_NULS` or `MATCH_FULS` can be used in a single call to initialize the set before adding or removing the specific characters.

The bits `MATCH_ADDC`, `MATCH_REMC` and `MATCH_INVC` can be used to add characters to the set, remove characters from the set or to invert character membership. The characters are taken from the character array pointed by the function argument `pch`.

If more than one bit of these is set then result is undefined. Current implementation checks these bits in a specific order, but later versions may change.

If none of these bits is set the value of the pointer `pch` is ignored.

It is no problem if a character is already in the set and is added or if it is not member of the set and is removed. Although it has no practical importance the array pointed by `pch` may contain a character many times.

```
void match_ModifySet(pMatchSets pMS,
                     char JokerCharacter,
                     int nChars,
                     unsigned char *pch,
                     int fAction
);
```

1.12.4 match

FUNCTION:

`match` checks if pszString matches the pattern pszPattern. pszPattern is a string containing joker characters. These are:

```
* matches one or more any character
# matches one or more digit
$ matches one or more alphanumeric character
 matches one or more alpha character
  (space) matches one or more spaces
? matches a single character
```

`~x` matches `x` even if `x` is pattern matching character or tilde

`x` matches character `x` unless it is a joker character

RETURN VALUE:

The function returns zero if no error occures and returns an error code in case some of the memory buffer does not have enough space. (Either pszBuffer or ParameterArray)

PARAMETERS:

`pszPattern` IN the pattern to match

–

`cbPattern` IN the number of characters in the pattern

–

`pszString` IN the string which is compared to the pattern

–

`cbString` IN the number of characters in the string

–

`ParameterArray` OUT is an uninitialized character pointer array. Upon return `ParameterArray[i]` points the string that matches the `i`-th joker character.

–

`pcbParameterArray` OUT is an unininitalized `unsigned long` array. Upon return `pcbParameterArray[i]` contains the length of the output parameter `ParameterArray[i]`.

–

`pszBuffer` OUT should point to a buffer. The size of the buffer should be specified by cbBufferSize. A size equal

`cbString`

is a safe size. The actual strings matching the joker characters will get into this buffer zero terminated one after the other:

–

`cArraySize` IN number of elements in the array `ParameterArray`

–

`cbBufferSize` IN size of the buffer pointed by pszBuffer

–

`fCase` IN pattern matching is performed case sensitive if this value if TRUE.

–

`iResult` OUT TRUE if `pszString` matches the pattern `pszPattern`. FALSE otherwise.

NOTE:

`pszPattern` and `pszString` are NOT changed.

If the function returns non-zero (error code) none of the output variables can be reliably used.

```
int match_match(char *pszPattern,
                unsigned long cbPattern,
                char *pszString,
                unsigned long cbString,
                char **ParameterArray,
                unsigned long *pcbParameterArray,
                char *pszBuffer,
                int cArraySize,
                int cbBufferSize,
                int fCase,
                pMatchSets pThisMatchSets,
                int *iResult
);
```

1.12.5 count

This function counts the number of jokers in the string and returns it. This function should be used to calculate the safe length of the pszBuffer given as a parameter to match.

```
int match_count(char *pszPattern,
                unsigned long cbPattern
);
```

1.12.6 parameter

This function takes a format string and a string array and copies the format string replacing `$0`, `$1` ... `$n` values with the appropriate string values given in the array pointed by ParameterArray.

RETURN VALUE:

The function returns zero if no error occures and returns an error code in case some of the memory buffer does not have enough space or invalid parameter is referenced.

PARAMETERS: `pszFormat` IN The format string containing the $i placeholders.

–

`cbFormat` IN The number of characters in the format string

–

`ParameterArray` IN string array so that `ParameterArray[i]` is to be inserted in place of the `$i` placeholders

–

`pcbParameterArray` IN array of `unsigned long` values. `pcbParameterArray[i]` gives the length of the i-th string parameter.

–

`pszBuffer` OUT buffer to put the result

–

`cArraySize` IN Number of parameters given in the ParameterArray

–

`pcbBufferSize` IN/OUT Available bytes in buffer pointed by `pszBuffer`. Upon return it contains the number of characters that were placed in the buffer.

–

NOTE:

If the function returns non-zero (error code) none of the output variables can be reliably used.

```
int match_parameter(char *pszFormat,
                    unsigned long cbFormat,
                    char **ParameterArray,
                    unsigned long *pcbParameterArray,
                    char *pszBuffer,
                    int cArraySize,
```

```
                        unsigned long *pcbBufferSize
);
```

1.12.7 size

Calculate the size of the output. The IN/OUT parameter `cbBufferSize` is increased by the number of needed characters.

The return value is zero if no error occured or the error code.

NOTE: cbBuffer size should be initialized to 0 if you want to get the size of the buffer needed.

```
int match_size(char *pszFormat,
               unsigned long cbFormat,
               unsigned long *pcbParameterArray,
               int cArraySize,
               int *cbBufferSize
);
```

1.13 sym.c

1.13.1 sym_NewSymbolTable()

This function creates a new symbol table. Later this symbol table should be used to store and retrieve symbol information.

```
SymbolTable sym_NewSymbolTable(
  void* (*memory_allocating_function)(size_t,void *),
  void *pMemorySegment
);
```

The second argument should point to the memory allocating function that the symbol table creation process should use. The last argument is an pointer to a memory segment which is passed to the memory allocation function. The actual arguments of the memory allocation function fits the allocation function from the package alloc. However the defintion is general enough to use any other function.

1.13.2 sym_FreeSymbolTable()

This function should be used to release the memory allocated for a symbol table. This function releases all the memory that was allocated during symbol table creation and during symbol insertion.

Note that the memory allocated outside the symbol table handling routines is not released. This means that it is the caller responsibility to relase all memory that holds the actual values associated with the symbols.

```
void sym_FreeSymbolTable(
  SymbolTable table,
```

```
  void (*memory_releasing_function)(void *,void *),
  void *pMemorySegment
);
```

1.13.3 sym_TraverseSymbolTable()

This function can be used to traverse through all the symbols stored in a symbol table. The function starts to go through the symbols and for each symbol calls the function `call_back_function`.

```
void sym_TraverseSymbolTable(
  SymbolTable table,
  void (*call_back_function)(char *SymbolName, void *SymbolValue, void *f),
  void *f
);
```

The first argument is the symbol table to traverse. The second argument is the function to be called for each symbol. This function gets three arguments. The first is a pointer to the symbol string. The second is the pointer to the symbol arguments. The third argument is a general pointer which is passed to the function `sym_TraverseSymbolTable`.

Note that the call back function gets the pointer to the symbol arguments and not the pointer to the pointer to the symbol arguments, and therefore call back function can not change the actual symbol value pointer.

1.13.4 sym_LookupSymbol()

This function should be used to search a symbol or to insert a new symbol.

```
void **sym_LookupSymbol(
  char *s,                  /* zero terminated string containing the symbol
  SymbolTable hashtable,    /* the symbol table
  int insert,               /* should a new empty symbol inserted, or return NULL in
  void* (*memory_allocating_function)(size_t, void *),
  void (*memory_releasing_function)(void *, void *),
  void *pMemorySegment
);
```

This function usually returns a pointer to the `void *` pointer which is supposed to point to the structure, which actually holds the parameters for the symbol. When a symbol is not found in the symbol table the parameter `insert` is used to decide what to do. If this parameter is zero the function returns `NULL`. If this parameter is 1 the function creates a new symbol and returns a pointer to the `void *` pointer associated with the symbol.

If a new symbol is to be inserted and the function returns `NULL` means that the memory allocation function has failed.

If the new symbol was created and a pointer to the `void *` pointer is returned the value of the pointer is `NULL`. In other words:

```
void **a;
```

```
a = sym_LookupSymbol(s,table,1,mymema,mymemr,p);

if( a == NULL )error("memory releasing error");
if( *a == NULL )error("symbol not found");
```

1.13.5 sym_DeleteSymbol()

This function should be used to delete a symbol from the symbol table

```
int sym_DeleteSymbol(
  char *s,                 /* zero terminated string containing the symbol
  SymbolTable hashtable,   /* the symbol table
  void (*memory_releasing_function)(void *, void *),
  void *pMemorySegment
);
```

This function searches the given symbol and if the symbol is found it deletes it from the symbol table. If the symbol was found in the symbol table the return value is zero. If the symbol was not found the return value is 1. This may be interpreted by the caller as an error or as a warning.

Note that this function only deletes the memory that was part of the symbol table. The memory allocated by the caller and handled via the pointer `value` usually returned by See ⟨undefined⟩ [sym_LookupSymbol()], page ⟨undefined⟩ should be released by the caller.

1.14 execute.c

This module contain the functions that execute the code resuled by the builder.

1.14.1 execute_GetCommandByName()

The op-code of a command can easily be identified, because `syntax.h` contains symbolic constant for it. This function can be used by external modules to get this opcode based on the name of the function. The argument `pszCommandName` should be the name of the command, for example `"ONERRORRESUMENEXT"`. The third argument is the hint for the function to help to find the value. It should always be the opcode of the command. The return value is the actual opcode of the command. For example:

```
i = execute_GetCommandByName(pEo,"ONERRORRESUMENEXT",CMD_ONERRORRESUMENEXT);
```

will return `CMD_ONERRORRESUMENEXT`.

Why is this function all about then?

The reason is that the external module may not be sure that the code `CMD_ONERRORRESUMENEXT` is the same when the external module is compiled and when it is loaded. External modules negotiate the interface version information with the calling interpreter, but the opcodes may silently changed from interpreter version to the next interpreter version and still supporting the same extension interface version.

When an external module needs to know the opcode of a command of the calling interpreter it first calls this function telling:

I<I need the code of the command ONERRORRESUMENEXT. I think that the code is CMD_ONERRORRESUMENEXT, but is it the real code?>

The argument `lCodeHint` is required only, because it speeds up search.

If there is no function found for the given name the returnvalue is zero.

```
long execute_GetCommandByName(pExecuteObject pEo,
                              char *pszCommandName,
                              long lCodeHint
);
```

1.14.2 execute_CopyCommandTable()

The command table is a huge table containing pointers to functions. For example the `CMD_LET`-th element of the table points to the function `COMMAND_LET` implementing the assignment command.

This table is usually treated as constant and is not moduified during run time. In case a module wants to reimplement a command it should alter this table. However the table is shared all concurrently running interpreter threads in a multi-thread variation of ScriptBasic.

To avoid altering the command table of an independent interpreter threadthe module wanting altering the command table should call this function. This function allocates memory for a new copy of the command table and copies the original constant value to this new place. After the copy is done the `ExecuteObject` will point to the copied command table and the extension is free to alter the table.

In case the function is called more than once for the same interpreter thread only the first time is effective. Later the function returns without creating superfluous copies of the command table.

```
int execute_CopyCommandTable(pExecuteObject pEo
);
```

1.14.3 execute_InitStructure()

```
int execute_InitStructure(pExecuteObject pEo,
                          pBuildObject pBo
);
```

1.14.4 execute_ReInitStructure()

This function should be used if a code is executed repeatedly. The first initialization call is See ⟨undefined⟩ [execute_InitStructure()], page ⟨undefined⟩ and consecutive executions should call this function.

```
int execute_ReInitStructure(pExecuteObject pEo,
                            pBuildObject pBo
);
```

1.14.5 execute_Execute_r()

This function executes a program fragment. The execution starts from the class variable `ProgramCounter`. This function is called from the See ⟨undefined⟩ [execute_Execute()], page ⟨undefined⟩ function which is the main entry point to the basic main program. This function is also called recursively from the function See ⟨undefined⟩ [execute_Evaluate()], page ⟨undefined⟩ when a user defined function is to be executed.

```
void execute_Execute_r(pExecuteObject pEo,
                       int *piErrorCode
);
```

1.14.6 execute_InitExecute()

```
void execute_InitExecute(pExecuteObject pEo,
                         int *piErrorCode
);
```

1.14.7 execute_FinishExecute()

```
void execute_FinishExecute(pExecuteObject pEo,
                           int *piErrorCode
);
```

1.14.8 execute_Execute()

This function was called from the basic `main` function. This function performs initialization that is needed before each execution of the code and calls See ⟨undefined⟩ [execute_Execute_r()], page ⟨undefined⟩ to perform the execution.

Note that See ⟨undefined⟩ [execute_Execute_r()], page ⟨undefined⟩ is recursively calls itself.

This function is obsolete and is not used anymore. This is kept in the source for the shake of old third party variations that may depend on this function.

Use of this function in new applications is discouraged.

```
void execute_Execute(pExecuteObject pEo,
                     int *piErrorCode
);
```

1.14.9 execute_ExecuteFunction()

This function is used by the embedding layer (aka `scriba_` functions) to execute a function. This function is not directly called by the execution of a ScriptBasic program. It may be used after the execution of the program by a special embeddign application that keeps the code and the global variables in memory and calls functions of the program.

The function takes `pEo` as the execution environment. `StartNode` should be the node where the sub or function is defined. `cArgs` should give the number of arguments. `pArgs`

should point to the argument array. `pResult` will point to the result. If `pResult` is `NULL` the result is dropped. Otherwise the result is a mortal variable.

Note that this code does not check the number of arguments you provide. There can be more arguments passed to the SUB than it has declared, therefore you can initialize the local variables of the sub. (You should know that arguments are local variables in ScriptBasic just as any other non-argument local variable.)

The arguments should be normal immortal variables. They are passed to the SUB by reference and in case they are modified the old variable is going to be released.

`piErrorCode` returns the error code of the execution which is zero in case of no error.

```
void execute_ExecuteFunction(pExecuteObject pEo,
                             unsigned long StartNode,
                             long cArgs,
                             pFixSizeMemoryObject *pArgs,
                             pFixSizeMemoryObject *pResult,
                             int *piErrorCode
);
```

1.14.10 execute_Evaluate()

This function evaluates an expression. You should not get confused! This is not syntax analysis, caring operator precedences and grouping by nested parentheses. That has already been done during syntax analysis. This code performs the code that was generated from an expression.

The result is usually a mortal memory value which is the final result of the expression. However this piece of code assumes that the caller is careful enough to handle the result as read only, and sometimes the return value is not mortal. In this case the return value is a memory object that a variable points to. Whenever the caller needs this value to perform an operation that does not alter the value it is OK. Duplicating the structure to create a mortal would be waste of time and memory. On the other hand sometimes operations modify their operands assuming that they are mortal values. They should be careful.

Operators are actually created in the directory `commands` and they use the macros defined in `command.h` (created by `headerer.pl` from `command.c`). They help to avoid pitfalls.

The argument `iArrayAccepted` tells the function whether an array as a result is accepted or not. If a whole array is accepted as a result of the expression evaluation the array is returned. If the array is not an acceptable result, then the first element of the array is retuned in case the result is an array. If the result is NOT an array this parameter has no effect.

```
pFixSizeMemoryObject execute_Evaluate(pExecuteObject pEo,
                                      unsigned long lExpressionRootNode,
                                      pMortalList pMyMortal,
                                      int *piErrorCode,
                                      int iArrayAccepted
);
```

1.14.11 execute_LeftValue()

This function evaluate a left value. A left value is a special expression that value can be assigned, and therefore they usually stand on the left side of the assignment operator. That is the reason for the name.

When an expression is evaluates a pointer to a memory object is returned. Whenever a left value is evaluated a pointer to the variable is returned. If any code assignes value to the variable pointed by the return value of this function it should release the memory object that the left value points currently.

```
pFixSizeMemoryObject *execute_LeftValue(pExecuteObject pEo,
                                        unsigned long lExpressionRootNode,
                                        pMortalList pMyMortal,
                                        int *piErrorCode,
                                        int iArrayAccepted
);
```

1.14.12 execute_EvaluateArray()

This function should be used to evaluate an array access to get the actual value. This is called by See ⟨undefined⟩ [execute_Evaluate()], page ⟨undefined⟩.

An array is stored in the expression as an operator with many operands. The first operand is a local or global variable, the rest of the operators are the indices.

Accessing a variable holding scalar value with array indices automatically converts the variable to array. Accessing an array variable without indices gets the "first" element of the array.

```
pFixSizeMemoryObject execute_EvaluateArray(pExecuteObject pEo,
                                           unsigned long lExpressionRootNode,
                                           pMortalList pMyMortal,
                                           int *piErrorCode
);
```

1.14.13 execute_EvaluateSarray()

This function should be used to evaluate an array access to get the actual value. This is called by See ⟨undefined⟩ [execute_Evaluate()], page ⟨undefined⟩.

An array is stored in the expression as an operator with many operands. The first operand is a local or global variable, the rest of the operators are the indices.

Associative arrays are normal arrays, only the access mode is different. When accessing an array using the fom `akey` then the access searches for the value `key` in the evenly indexed elements of the array and gives the next index element of the array. This if

```
a[0] = "kakukk"
a[1] = "birka"
a[2] = "kurta"
a[3] = "mamus"
```

then `a"kakukk"` is "birka". `a"birka"` is `undef`. `a"kurta"` is "mamus".

```
pFixSizeMemoryObject execute_EvaluateSarray(pExecuteObject pEo,
                                            unsigned long lExpressionRootNode,
                                            pMortalList pMyMortal,
                                            int *piErrorCode
);
```

1.14.14 execute_LeftValueArray()

This function evaluates an array access left value. This function is also called by See ⟨undefined⟩ [execute_EvaluateArray()], page ⟨undefined⟩ and the result pointer is dereferenced.

```
pFixSizeMemoryObject *execute_LeftValueArray(pExecuteObject pEo,
                                             unsigned long lExpressionRootNode,
                                             pMortalList pMyMortal,
                                             int *piErrorCode
);
```

1.14.15 execute_LeftValueSarray()

This function evaluates an associative array access left value. This function is also called by See ⟨undefined⟩ [execute_EvaluateSarray()], page ⟨undefined⟩ and the result pointer is dereferenced.

```
pFixSizeMemoryObject *execute_LeftValueSarray(pExecuteObject pEo,
                                              unsigned long lExpressionRootNode,
                                              pMortalList pMyMortal,
                                              int *piErrorCode
);
```

1.14.16 execute_Convert2String()

This functionconverts a variable to string. When the variable is already a string then it returns the pointer to the variable. When the variable is long or double `sprintf` is used to convert the number to string.

When the conversion from number to string is done the result is always a newly allocated mortal. In other words this conversion routine is safe, not modifying the argument memory object.

```
pFixSizeMemoryObject execute_Convert2String(pExecuteObject pEo,
                                            pFixSizeMemoryObject pVar,
                                            pMortalList pMyMortal
);
```

1.14.17 execute_Convert2Long()

This function should be used to convert a variable to long. The conversion is usually done in place. However strings can not be converted into long in place, because they have different size. In such a case a new variable is created. If the mortal list `pMyMortal` is `NULL`

then the new variable in not mortal. In such a case care should be taken to release the original variable.

Usually there is a mortal list and a new mortal variable is generated. In such a case the original value is also a mortal and is automatically released after the command executing the conversion is finished.

Note that strings are converted to long in two steps. The first step converts the string to `double` and then this value is converted to long in-place.

```
pFixSizeMemoryObject execute_Convert2Long(pExecuteObject pEo,
                                          pFixSizeMemoryObject pVar,
                                          pMortalList pMyMortal
);
```

1.14.18 execute_Convert2LongS()

This is the safe version of the conversion function See ⟨undefined⟩ [execute_Convert2Long()], page ⟨undefined⟩.

This function ALWAYS create a new variable and does NOT convert a double to long in place. This function is called by the extensions, because extensions tend to be more laisy regarding conversion and many converts arguments in place and thus introduce side effect.

To solve this problem we have introduced this function and have set the support table to point to this function.

```
pFixSizeMemoryObject execute_Convert2LongS(pExecuteObject pEo,
                                           pFixSizeMemoryObject pVar,
                                           pMortalList pMyMortal
);
```

1.14.19 execute_Convert2Double()

This function should be used to convert a variable to double. The conversion is usually done in place. However strings can not be converted into double in place, because they have different size. In such a case a new variable is created. If the mortal list is `NULL` then the new variable in not mortal. In such a case care should be taken to release the original variable.

Usually there is a mortal list and a new mortal variable is generated. In such a case the original value is also a mortal and is automatically released after the command executing the conversion is finished.

```
pFixSizeMemoryObject execute_Convert2Double(pExecuteObject pEo,
                                            pFixSizeMemoryObject pVar,
                                            pMortalList pMyMortal
);
```

1.14.20 execute_Convert2DoubleS()

This is the safe version of the conversion function See ⟨undefined⟩ [execute_Convert2Double()], page ⟨undefined⟩.

This function ALWAYS create a new variable and does NOT convert a long to double in place. This function is called by the extensions, because extensions tend to be more laisy regarding conversion and many converts arguments in place and thus introduce side effect.

To solve this problem we have introduced this function and have set the support table to point to this function.

```
pFixSizeMemoryObject execute_Convert2DoubleS(pExecuteObject pEo,
                                             pFixSizeMemoryObject pVar,
                                             pMortalList pMyMortal
);
```

1.14.21 execute_Convert2Numeric()

This function should be used to convert a variable to numeric type.

The conversion results a double or long variable. If the source variable was already a long or double the function does nothing but results the source variable.

`undef` is converted to long zero.

The function calls See ⟨undefined⟩ [execute_Convert2Long], page ⟨undefined⟩ and See ⟨undefined⟩ [execute_Convert2Double], page ⟨undefined⟩ thus all other parameters are treated according to that.

```
pFixSizeMemoryObject execute_Convert2Numeric(pExecuteObject pEo,
                                             pFixSizeMemoryObject pVar,
                                             pMortalList pMyMortal
);
```

1.14.22 execute_Dereference()

This function recursively follows variable references and returns the original variable that was referenced by the original variable.

A reference variable is a special variable that does not hold value itself but rather a pointer to another variable. Such reference variables are used when arguments are passed by reference to BASIC subroutines.

Calling this function the caller can get the original variable and the value of the original variable rather than a reference.

```
pFixSizeMemoryObject execute_Dereference(pExecuteObject pEo,
                                         pFixSizeMemoryObject p,
                                         int *piErrorCode
);
```

See also See ⟨undefined⟩ [execute_DereferenceS()], page ⟨undefined⟩.

1.14.23 execute_DereferenceS()

This function does the same as See ⟨undefined⟩ [execute_Dereference()], page ⟨undefined⟩ except that it has different arguments fitted to support external modules and `besXXX` macros.

```
int execute_DereferenceS(unsigned long refcount,
                         pFixSizeMemoryObject *p
);
```

See also See ⟨undefined⟩ [execute_Dereference()], page ⟨undefined⟩.

If the argument is referencing an `undef` value then this function converts the argument to be a real `NULL` to allow external modules to compare `besDEREFERENCE`d variables against `NULL`.

The subroutine is also error prone handling `NULL` pointer as argument, though it should never be happen if the external module programmer uses the macro `besDEREFERENCE`.

1.14.24 execute_GetDoubleValue()

Use this function whenever you want to access the **value** of a variable as a `double`. Formerly ScriptBasic in such situation converted the variable to double calling See ⟨undefined⟩ [execute_Convert2Double()], page ⟨undefined⟩ and then used the macro `DOUBLEVALUE`. This method is faster because this does not create a new mortal variable but returns directly the double value.

The macro `GETDOUBLEVALUE` can be used to call this function with the default execution environment variable `pEo`

Note however that the macro `GETDOUBLEVALUE` and `DOUBLEVALUE` are not interchangeable. `GETDOUBLEVALUE` is returnig a `double` while `DOUBLEVALUE` is a left value available to store a `double`.

```
double execute_GetDoubleValue(pExecuteObject pEo,
                              pFixSizeMemoryObject pVar
);
```

1.14.25 execute_GetLongValue()

Use this function whenever you want to access the **value** of a variable as a `long`. Formerly ScriptBasic in such situation converted the variable to long calling See ⟨undefined⟩ [execute_Convert2Long()], page ⟨undefined⟩ and then used the macro `LONGVALUE`. This method is faster because this does not create a new mortal variable but returns directly the long value.

The macro `GETLONGVALUE` can be used to call this function with the default execution environment variable `pEo`

Note however that the macro `GETLONGVALUE` and `LONGVALUE` are not interchangeable. `GETLONGVALUE` is returnig a `long` while `LONGVALUE` is a left value available to store a `long`.

```
long execute_GetLongValue(pExecuteObject pEo,
                          pFixSizeMemoryObject pVar
);
```

Please also note that the result of converting a string variable to LONG and then accessing its longvalue may not result the same number as calling this function. The reason is that conversion of a string to a LONG variable is done in two steps. First it converts the

string to a `double` and then it rounds the `double` value to `long`. On the other hand this function converts a string diretly to `long`.

For example the string `"3.7"` becomes 4 when converted to long and 3 when getting the value as a long.

1.14.26 execute_IsStringInteger()

This function should be used to check a string before converting it to numeric value. If the string contains only digits it should be converted to `long`. If the string contains other characters then it should be converted to double. This function decides what characters the string contains.

```
int execute_IsStringInteger(pFixSizeMemoryObject pVar
);
```

1.14.27 execute_IsInteger()

This function checks that a variable being long, double or string can be converted to long without loosing information.

```
int execute_IsInteger(pFixSizeMemoryObject pVar
);
```

1.15 dynlolib.c

The Dynamic Load Libraries are handled different on all operating systems. This file implements a common functional base handling the DLLs for ScriptBasic. All other modules of ScriptBasic that want to use DLLs should call only the functions implemented in this file.

=toc

1.15.1 dynlolib_LoadLibrary

This function loads a library and returns a pointer that can be used in other functions referencing the loaded library.

```
void *dynlolib_LoadLibrary(
  char *pszLibraryFile
);
```

The argument `pszLibraryFile` is the ZCHAR file name.

The file name is either absolute or relative. When a relative file name is specified the directories searched may be different on different operating systems.

1.15.2 dynlolib_FreeLibrary

This function releases the library that was loaded before using See ⟨undefined⟩ [dynlolib_LoadLibrary], page ⟨undefined⟩

```
void dynlolib_FreeLibrary(
  void *pLibrary
);
```

The argument `pLibrary` is the pointer, which was returned by the function See ⟨undefined⟩ [dynlolib_LoadLibrary], page ⟨undefined⟩

1.15.3 dynlolib_GetFunctionByName

This function can be used to get the entry point of a function of a loaded module specifying the name of the function.

```
void *dynlolib_GetFunctionByName(
  void *pLibrary,
  char *pszFunctionName
);
```

The argument `pLibrary` is the pointer, which was returned by the function See ⟨undefined⟩ [dynlolib_LoadLibrary], page ⟨undefined⟩

The argument `pszFunctionName` is the ZCAR function name.

1.16 conftree.c

1.16.1 cft_init()

Before calling any other configuration handling function the caller has to prepare a `tConfigTree` structure. To do this it has to call this function.

The first argument has to point to an allocated and uninitialized `tConfigTree` structure. The second argument has to point to a memory allocating function. The third argument has to point to the memory releasing function that is capable releasing the memory allocated by the memory allocating function.

The argument `pMemorySegment` should be the segment pointer to the memory handling functions. All memory allocation will be performed calling the `memory_allocating_function` and passing the `pMemorySegment` pointer as second argument to it. All memory releasing will be done via the function `memory_releasing_function` passing `pMemorySegment` pointer as second argument. This lets the caller to use sophisticated memory handling architecture.

On the other hand for the simple use all these three arguments can be `NULL`. In this case the configuration management system will use its own memory allocating and releasing function that simply uses `malloc` and `free`. In this case `pMemorySegment` is ignored.

For a ready made module that delivers more features see the alloc module of the ScriptBasic project at `http://scriptbasic.com`

```
int cft_init(ptConfigTree pCT,
             void *(*memory_allocating_function)(size_t, void *),
             void (*memory_releasing_function)(void *, void *),
             void *pMemorySegment
```

```
);
```

Note that suggested convention is to use the '.' character as separator for hierarchical key structures, but this is only a suggestion. In other words the module writers advice is to use `key.subkey.subsubkey` as key string for hierarchical strings. On the other hand you can use any character as separator except the zero character and except the characters that are used as key characters. You can write

```
key\subkey\subsubkey
```

if you are a windows geek. To do this you have to change the character saying

```
pCT->TC = '\\';
```

after calling the initialization function. You can change this character any time, this character is not used in the configuration structure. The only point is that you have to use the actual character when you have changed it. The best practice is to use the dot ever.

1.16.2 cft_GetConfigFileName()

This function tries to locate the configuration file. The working of this function is system dependant. There are two different implementations: one for UNIX and one for Win32.

WIN32

On Win32 systems the function tries to read the system registry. The value of the key given in the argument `env` is used and returned as the config file name. For example if the argument `env` is `Software\myprog\conf` then the registry value of the key `HKEY_LOCAL_MACHINE\Software\myprog\conf` will be returned as configuration file name. The program does not check that the file really exists. It only checks that the registry key exists, it is a string and has some value.

If the registry key does not exists the program tries to locate the system directory getting the environment variable `windir`, then `systemroot` and finally taking `c:\WINDOWS`. The argument `DefaultFileName` is appended to the directory name and is returned.

UNIX

On UNIX it is more simple. The environment variable `env` is used as a file name. If this does not exists the `DefaultFileName` is used and returned.

BOTH

The return value of the function is zero if no error has happened. A pointer to the resulting file name is returned in the variable `ppszConfigFile`. The space to hold the resulting file name is allocated via the allocation function given by the `tConfigTree` structure pointed by `pCT`.

```
int cft_GetConfigFileName(ptConfigTree pCT,
                          char **ppszConfigFile,
                          char *env,/* environment variable or registry key on win3
                          char *DefaultFileName
);
```

This function is `static` and can not be called from outside of this module.

1.16.3 cft_start()

When writing real applications you usually want to call this function. This function initializes the `tConfigTree` structure pointed by `pCT`, searches for the configuration file and reads it.

When trying to allocate the configuration file the static internal function See ⟨undefined⟩ [GetConfigFileName], page ⟨undefined⟩ is used.

The argument `Envir` is the registry key under `HKLM`, eg `Software\Myprog\conf` under Win32 or the environment variable to look for the configuration file name. The argument `pszDefaultFileName` is the file name searched on WIN32 in the system directories or the full path to the default configuration file nam eunder UNIX. The argument `pszForcedFileName` can overrride the file name search or has to be `NULL` to let the reader search the environment and registry for file name.

```
int cft_start(ptConfigTree pCT,
              void *(*memory_allocating_function)(size_t, void *),
              void (*memory_releasing_function)(void *, void *),
              void *pMemorySegment,
              char *Envir,
              char *pszDefaultFileName,
              char *pszForcedFileName
);
```

1.16.4 strmyeq()

This is an internal `static` function that compares two strings and returns true iff they are equal. The string terminator is the usual zero character or the dot. Both are legal terminators for this functions and their difference in the compared strings is not treated as difference in the result. If one string is terminated by zero character and the other is terminated by a dot but they are the same in any other character then the return value is true.

This function is used find a sub-key when the caller has specified a dot separated hierarchical key.

Note that the dot is only a convention and the default value for the separator and the caller has

```
/**/
static int strmyeq(ptConfigTree pCT,char *a, char *b)
```

This function is `static` and can not be called from outside of this module.

1.16.5 cft_FindNode()

Find a node starting from the start node `lStartNode` and searching for the `key`.

The function returns zero if the key is not found in the configuration information tree `pCT` or returns the node id of the key. This node can either be an internal node or leaf.

Note that the string `key` may contain dot characters. In this case the key is searched down in the configuration tree. (You can set the separator character different from the dot character.)

```
CFT_NODE cft_FindNode(ptConfigTree pCT,
                      CFT_NODE lStartNode,
                      char *key
);
```

You need this function when you want to iterate over the sub-keys of a node. You get the node id for the key and then you can call See ⟨undefined⟩ [cft_EnumFirst], page ⟨undefined⟩ to start the loop and then See ⟨undefined⟩ [cft_EnumNext], page ⟨undefined⟩ to iterate the loop over the sub-keys.

If you just want to get the value of a single key you can call the function See ⟨undefined⟩ [cft_GetEx], page ⟨undefined⟩ that uses this function.

1.16.6 cft_GetEx()

Get the value associated with the key `key` from the configuration structure `pCT`, or get the values of a node.

The arguments:

`pCT` the configuration information searched.

`key` the key that we search the value for, or NULL if we already know the node id where the needed information is.

`plNodeId` the id of the node that we need information from. If the key argumentum is not NULL then this argument is overwritten with the node id associated with the key. If the argument key is NULL this argument should specify the id of the node we need information from. If the node id is not needed upon return this argument may point to NULL.

`ppszValue` will return a pointer to a constant ZCHAR string if the value associated with `key` is string. If the argument is `NULL` then the function ignore this argument.

`plValue` will return a `long` if the value associated with `key` is integer. If the argument is `NULL` then the function ignore this argument.

`pdValue` will return a `double` if the value associated with `key` is a real number. If the argument is `NULL` then the function ignore this argument.

`type` will return the type of the key. This can be

`CFT_NODE_BRANCH` if the key is associated with a subtree.

`CFT_TYPE_STRING` if the key is associated with a string

`CFT_TYPE_INTEGER` if the key is associated with an integer number

`CFT_TYPE_REAL` if the key is associated with a real number

This argument can also be NULL if the caller is not interested in the type of the value.

Note that any of `ppszValue`, `plValue`, `pdValue` can point to a variable or to `NULL` in case the caller does not need the actual value.

```
int cft_GetEx(ptConfigTree pCT,
              char *key,
              CFT_NODE *plNodeId,
              char **ppszValue,
              long *plValue,
              double *pdValue,
              int *type
);
```

The function returns `CFT_ERROR_SUCCESS` if no error happens. The value `CFT_ERROR_SUCCESS` is zero.

If an error happens the error code is returned. These error codes are:

`CFT_ERROR_NOT_FOUND` the key is not present in the table, and `*plNodeId` will also be set to zero.

`CFT_ERROR_NOTYPE` the key is found but has a type that can not be returned, because the caller passed NULL as storage location. In this case the type of the configuration information is probably wrong.

1.16.7 cft_GetString()

This is the simplest interface function to retrieve a configuration string. This assumes that you exactly know the name of the key and you are sure that the value is a string. The function returns the pointer to the constant string or returns NULL if the configuration key is not present in the tree or the value is not a string.

The use of this function is not recommended. This function is present in this package to ease porting of programs that use simpler configuration information management software.

```
char *cft_GetString(ptConfigTree pCT,
                    char *key
);
```

This function calls See ⟨undefined⟩ [cft_GetEx], page ⟨undefined⟩.

1.16.8 cft_EnumFirst()

Whenever you need to enumerate the sub-keys of a key you have to get the node associated with the key (see See ⟨undefined⟩ [cft_GetEx], page ⟨undefined⟩ or See ⟨undefined⟩ [cft_FindNode], page ⟨undefined⟩). When you have the node associated with the key you can get the node of the first sub-key calling this function.

The function needs the node id `lNodeId` of the key for which we need to enumerate the sub keys and returns the node id of the first sub key.

If the key is associated with a leaf node the function returns zero.

If the key is associated with a branch node that has no sub-keys the function returns zero.

```
CFT_NODE cft_EnumFirst(ptConfigTree pCT,
                       CFT_NODE lNodeId
);
```

1.16.9 cft_EnumNext()

Whenever you need to enumerate the sub-keys of a key you have to get the node associated with the key (see See ⟨undefined⟩ [cft_GetEx], page ⟨undefined⟩ or See ⟨undefined⟩ [cft_FindNode], page ⟨undefined⟩). When you have the node associated with the key you can get the node of the first sub-key calling the function See ⟨undefined⟩ [cft_EnumFirst], page ⟨undefined⟩. Later on you can enumerate the sub keys stepping from node to node calling this function.

The function needs the node id `lNodeId` returned by See ⟨undefined⟩ [cft_EnumFirst], page ⟨undefined⟩ or by previous call of this function.

The function returns the node id of the next sub key.

If the enumeration has ended, in other words there is no next sub-key the function returns zero.

```
long cft_EnumNext(ptConfigTree pCT,
                  long lNodeId
);
```

1.16.10 cft_GetKey()

This function returns a pointer to the constant zchar string that holds the key of the node defined by the id `lNodeId`.

```
char *cft_GetKey(ptConfigTree pCT,
                 CFT_NODE lNodeId
);
```

1.16.11 cft_ReadConfig()

```
int cft_ReadConfig(ptConfigTree pCT,
                   char *pszFileName
);
```

1.16.12 cft_WriteConfig()

```
int cft_WriteConfig(ptConfigTree pCT,
                    char *pszFileName
);
```

1.16.13 cft_DropConfig()

```
void cft_DropConfig(ptConfigTree pCT
);
```

1.17 filesys.c

=abstract The file `filesys.h` contains file handling primitive functions. The reason for this module is to have all system specific file handling functions to be separated in a single file. All other modules use these functions that behave the same on Win32 platform as well as on UNIX. =end These functions are to be used by other parts of the program. They implement system specific operations, and other levels need not care about these system specific stuff.

The function names are prefixed usually with `file_`, some are prefixed with `sys_`.

=toc

1.17.1 file_fopen

This is same as fopen.

VMS has some specialities when writing a file.

```
FILE *file_fopen(
  char *pszFileName,
  char *pszOpenMode
);
```

1.17.2 file_fclose

This is same as fclose. Nothing special. This is just a placeholder.

```
void file_fclose(FILE *fp
);
```

1.17.3 file_size

```
long file_size(char *pszFileName
);
```

1.17.4 file_time_accessed

```
long file_time_accessed(char *pszFileName
);
```

1.17.5 file_time_modified

```
long file_time_modified(char *pszFileName
);
```

1.17.6 file_time_created

```
long file_time_created(char *pszFileName
);
```

1.17.7 file_isdir

```
int file_isdir(char *pszFileName
);
```

1.17.8 file_isreg

```
int file_isreg(char *pszFileName
);
```

1.17.9 file_exists

```
int file_exists(char *pszFileName
);
```

1.17.10 file_truncate

It return 0 on success and -1 on error.

```
int file_truncate(FILE *fp,
                  long lNewFileSize
);
```

1.17.11 file_fgetc

Nothing special, it is just a placeholder.

```
int file_fgetc(FILE *fp
);
```

1.17.12 file_ferror

Nothing special, it is just a placeholder.

```
int file_ferror(FILE *fp
);
```

1.17.13 file_fread

Nothing special, it is just a placeholder.

```
int file_fread(char *buf,
               int size,
               int count,
               FILE *fp
);
```

1.17.14 file_fwrite

Nothing special, it is just a placeholder.

```
int file_fwrite(char *buf,
                int size,
                int count,
                FILE *fp
);
```

1.17.15 file_fputc

Nothing special, it is just a placeholder.

```
int file_fputc(int c, FILE *fp
);
```

1.17.16 file_setmode

Nothing special, it is just a placeholder. On UNIX this is doing nothing transparently.

```
void file_setmode(FILE *fp,
                  int mode
);
```

1.17.17 file_binmode

```
void file_binmode(FILE *fp
);
```

1.17.18 file_textmode

```
void file_textmode(FILE *fp
);
```

1.17.19 file_flock

```
int file_flock(FILE *fp,
               int iLockType
);
```

1.17.20 file_lock

```
int file_lock(FILE *fp,
              int iLockType,
              long lStart,
              long lLength
);
```

1.17.21 file_feof

Nothing special, it is just a placeholder.

```
int file_feof(FILE *fp
);
```

1.17.22 file_mkdir

This is the usual UNIX mkdir function. The difference is that the access code is always 0777 on UNIX which means that the user, group and others can read, write and execute the directory. If the permission needed is different from that you have to call the `file_chmod` function as soon as it becomes available.

The argument of the function is the name of the desired directory.

```
int file_mkdir(char *pszDirectoryName
);
```

1.17.23 file_rmdir

This is the usual UNIX rmdir function.

The argument of the function is the name of the directory to be deleted.

```
int file_rmdir(char *pszDirectoryName
);
```

1.17.24 file_remove

Nothing special, it is just a placeholder. This function performs the UNIX `remove` functionality. This function also exists under WIN32, therefore this function is only a placeholder.

```
int file_remove(char *pszFileName
);
```

1.17.25 file_deltree

```
int file_deltree(char *pszDirectoryName
);
```

1.17.26 file_MakeDirectory

This function is a bit out of the line of the other functions in this module. This function uses the `file_mkdir` function to create a directory. The difference is that this function tries to create a directory recursively. For example you can create the directory

`/usr/bin/scriba`

with a simple call and the function will create the directories `/usr` if it did not exist, then `/usr/bin` and finally `/usr/bin/scriba` The function fails if the directory can not be

created because of access restrictions or because the directory path or a sub path already exists, and is not a directory.

The argument of the function is the name of the desired directory.

The function alters the argument replacing each \ character to /

The argument may end with / since v1.0b30

If the argument is a Windows full path including the drive letter, like 'C:' the function tries to create the directory 'C:', which fails, but ignores this error because only the last creation in the line down the directory path is significant.

In case of error, the argument may totally be destroyed.

```
int file_MakeDirectory(char *pszDirectoryName
);
```

1.17.27 file_opendir

This function implements the `opendir` function of UNIX. The difference between this implementation and the UNIX version is that this implementation requires a `DIR` structure to be passed as an argument. The reason for this is that the Windows system calls do not allocate memory and pass return values in structures allocated by the caller. Because we did not want to implement memory allocation in these routines we followed the Windows like way.

The first argument `pszDirectoryName` is a ZCAR directory name to be scanned. The second argument is an allocated `DIR` structure that has to be valid until the `file_closedir` is called.

The second parameter under UNIX is not used. However to be safe and portable to Win32 the parameter should be handled with care.

```
DIR *file_opendir(char *pszDirectoryName,
                  tDIR *pDirectory
);
```

1.17.28 file_readdir

This function is the implementation of the UNIX `readdir`

```
struct dirent *file_readdir(DIR *pDirectory
);
```

1.17.29 file_closedir

```
void file_closedir(DIR *pDirectory
);
```

1.17.30 file_sleep

```
void sys_sleep(long lSeconds
);
```

1.17.31 file_curdir

The first argument should point to a buffer having space for at least `cbBuffer` characters. The function will copy the name of the current directory into this buffer.

Return value is zero on success. If the current directory can not be retrieved or the buffer is too short the return value is -1.

```
int file_curdir(char *Buffer,
                unsigned long cbBuffer
);
```

1.17.32 file_chdir

```
int file_chdir(char *Buffer
);
```

1.17.33 file_chown

This function implements the chown command of the UNIX operating system on UNIX and Windows NT. The first argument is the ZCHAR terminated file name. No wild card characters are allowed.

The second argument is the name of the desired new user. The function sets the owner of the file to the specified user, and returns zero if the setting was succesful. If the setting fails the function returns an error code. The error codes are:

COMMAND_ERROR_CHOWN_NOT_SUPPORTED COMMAND_ERROR_CHOWN_INVALID_USE
COMMAND_ERROR_CHOWN_SET_OWNER

```
int file_chown(char *pszFile,
               char *pszOwner
);
```

1.17.34 file_getowner

```
int file_getowner(char *pszFileName,
                  char *pszOwnerBuffer,
                  long cbOwnerBuffer
);
```

1.17.35 file_SetCreateTime

Note that this time value does not exist on UNIX and therefore calling this function under UNIX result error.

The argument to the function is the file name and the desired time in number of seconds since the epoch. (January 1, 1970. 00:00)

If the time was set the return value is zero. If there is an error the return value is the error code.

```
int file_SetCreateTime(char *pszFile,
                       long lTime
);
```

1.17.36 file_SetModifyTime

The argument to the function is the file name and the desired time in number of seconds since the epoch. (January 1, 1970. 00:00)

If the time was set the return value is zero. If there is an error the return value is the error code.

```
int file_SetModifyTime(char *pszFile,
                       long lTime
);
```

1.17.37 file_SetAccessTime

The argument to the function is the file name and the desired time in number of seconds since the epoch. (January 1, 1970. 00:00)

If the time was set the return value is zero. If there is an error the return value is the error code.

```
int file_SetAccessTime(char *pszFile,
                       long lTime
);
```

1.17.38 file_gethostname

This function gets the name of the host that runs the program. The result of the function is positive if no TCP/IP protocol is available on the machine or some error occured.

In case of success the return value is zero.

```
int file_gethostname(char *pszBuffer,
                     long cbBuffer
);
```

The first argument should point to the character buffer, and the second argument should hold the size of the buffer in bytes.

1.17.39 file_gethost

This function gets the `struct hostent` entry for the given address. The address can be given as a FQDN or as an IP octet tuple, like www.digital.com or 16.193.48.55

Optionally the address may contain a port number separated by : from the name or the IP number. The port number is simply ignored.

```
int file_gethost(char *pszBuffer,
                 struct hostent *pHost
);
```

`pszBuffer` should hold the name or the address of the target machine. This buffer is not altered during the function.

`pHost` should point to a buffer ready to hold the hostent information.

Note that the structure `hostent` contains pointers outside the structre. Those pointers are copied verbatim thus they point to the original content as returned by the underlying socket layer. This means that the values the `hostent` structure points to should not be freed, altered and the values needed later should be copied as soon as possible into a safe location before any other socket call is done.

1.17.40 file_tcpconnect

This function tries to connect to the remote port of a remote server. The first argument of the function should be a pointer to `SOCKET` variable as defined in `filesys.h` or in the Windows header files. The second argument is a string that contains the name of the remote host, or the IP number of the remote host and the desired port number following the name separated by a colon. For example `index.hu:80` tries to connect to the http port of the server `index.hu`. You can also write `16.192.80.33:80` to get a connection. The function automatically recognizes IP numbers and host names. The socket is created automatically calling the system function `socket`.

If the function successfully connected to the remote server the return value is zero. Otherwise the return value is the error code.

```
int file_tcpconnect(SOCKET *sClient,
                    char *pszRemoteSocket
);
```

1.17.41 file_tcpsend

```
int file_tcpsend(SOCKET sClient,
                 char *pszBuffer,
                 long cbBuffer,
                 int iFlags
);
```

1.17.42 file_tcprecv

```
int file_tcprecv(SOCKET sClient,
                 char *pszBuffer,
                 long cbBuffer,
                 int iFlags
);
```

1.17.43 file_tcpclose

```
int file_tcpclose(SOCKET sClient
);
```

1.17.44 file_killproc

This function kills a process identified by the process ID (PID).

If the process is killed successfully the return value is zero, otherwise a positive value.

```
int file_killproc(long pid
);
```

1.17.45 file_fcrypt

This function implements the password encryption algorithm using the DES function. The first argument is the clear text password, the second argument is the two character salt value. This need not be zero terminated. The third argument should point to a 13 characters char array to get the encoded password. `buff[13]` will contain the terminating zchar upon return.

```
char *file_fcrypt(char *buf, char *salt, char *buff
);
```

1.17.46 file_CreateProcess

This function creates a new process using the argument as command line. The function does NOT wait the new process to be finished but returns the pid of the new process.

If the new process can not be started the return value is zero.

The success of the new process however can not be determined by the return value. On UNIX this value is generated by the fork system call and it still may fail to replace the executeable image calling `exevp`. By that time the new program creation is already in the newprocess and is not able to send back any error information to the caller.

The caller of this function should also check other outputs of the created process that of the pid is returned. For example if the `execv` call failed the process exit code is `1`. This is usually an error information of a process.

```
long file_CreateProcess(char *pszCommandLine
);
```

1.17.47 file_CreateProcessEx

This function starts a new process and starts to wait for the process. The caller can specify a timeout period in seconds until the function waits.

When the process terminates or the timeout period is over the function returns.

```
int file_CreateProcessEx(char *pszCommandLine,
                         long lTimeOut,
                         unsigned long *plPid,
                         unsigned long *plExitCode
);
```

Arguments:

`pszCommandLine` the command to execute

`lTimeOut` the maximum number of seconds to wait for the process to finish. If this is zero the function will not wait for the process. If the value is `-1` the function wait without limit until the created process finishes.

`plPid` pointer to variable where the PID of the new process is placed. This parameter can be `NULL`. If the function returns after the new process has terminated this value is more or less useless. However this parameter can be used to kill processes that reach the timeout period and do not terminate.

`plExitCode` pointer to a variable where the exit code of the new process is placed. If the process is still running when the function returns this parameter is unaltered.

The return value indicates the success of the execution of the new process:

`FILESYSE_SUCCESS` The process was started and terminated within the specified timeout period.

`FILESYSE_NOTSTARTED` The function could not start the new process. (not used under UNIX)

`FILESYSE_TIMEOUT` The process was started but did not finish during the timeout period.

`FILESYSE_NOCODE` The process was started and finished within the timeout period but it was not possible to retrieve the exit code.

Note that the behaviour of this function is slightly different on Windows NT and on UNIX. On Windows NT the function will return `FILESYSE_NOTSTARTED` when the new process can not be started. Under UNIX the process performs a `fork()` and then an `execv`. The `fork()` does not return an error value. When the `execvp` fails it is already in the new process and can not return an error code. It exists using the exit code 1. This may not be distinguished from the program started and returning an exit code 1.

1.17.48 file_waitpid

This function checks if a process identified by the process ID (PID) is still running.

If the process is live the return value is zero (FALSE), otherwise a positive value (TRUE) is returned and the second parameter contains the exited process's final status.

```
int file_waitpid(long pid,
                 unsigned long *plExitCode
);
```

1.18 modumana.c

This file contains all the functions that handle external module management.

Note that all function names are prepended by `modu_`

1.18.1 modu_Init

This function allocates memory for the external module interface table and initializes the function pointers.

If the interface already exists and the function is called again it just silently returns.

The second argument can be zero or 1. The normal operation is zero. If `iForce` is true the function sets each function pointer to its initial value even if an initialization has already occured before.

This can be used in a rare case when a module modifies the interface table and want to reinitialize it to the original value. Be carefule with such constructions.

```
int modu_Init(pExecuteObject pEo,
              int iForce
);
```

1.18.2 modu_Preload

```
int modu_Preload(pExecuteObject pEo
);
```

1.18.3 modu_GetModuleFunctionByName

This function gets the entrypoint of a module function. This module can either be statically or dynamically linked to ScriptBasic. This function is one level higher than See ⟨undefined⟩ [GetStaticFunctionByName], page ⟨undefined⟩ or See ⟨undefined⟩ [dynlolib_GetFunctionByName], page ⟨undefined⟩. The first argument to this function is not the module handle as returned by See ⟨undefined⟩ [dynlolib_LoadLibrary], page ⟨undefined⟩ but rather the pointer to the module description structure that holds other information on the modula. Namely the information that the module is loaded from dll or so, or if the module is linked to the interpreter static.

```
void *modu_GetModuleFunctionByName(
  pModule pThisModule,
  char *pszFunctionName
);
```

1.18.4 modu_GetStaticFunctionByName

Get the entry point of a function that was linked to the ScriptBasic environment statically.

This is the counterpart of the function `dynlolib_GetFunctionByName` for functions in library linked static. This function searches the `SLFST` table for the named function and returns the entry point or `NULL` if there is no functions with the given name defined.

```
void *modu_GetStaticFunctionByName(
  void *pLibrary,
  char *pszFunctionName
);
```

1.18.5 modu_LoadModule

This function loads a module and returns the module pointer to in the argument `pThisModule`. If the module is already loaded it just returns the module pointer.

When the function is called first time for a module it loads the module, calls the version negotiation function and the module initializer.

If module file name given in the argument `pszLibrary` file name is an absolute file name this is used as it is. Otherwise the different configured module directories are seached for the module file, and the operating system specific extension is also appended to the file name automatically.

If the caller does not need the pointer to the module the argument `pThisModule` can be `NULL`.

```
int modu_LoadModule(pExecuteObject pEo,
                    char *pszLibraryFile,
                    pModule **pThisModule
);
```

1.18.6 modu_GetFunctionByName

This function can be called to get the entry point of a function from an external module. If the module was not loaded yet it is automatically loaded.

```
int modu_GetFunctionByName(pExecuteObject pEo,
                           char *pszLibraryFile,
                           char *pszFunctionName,
                           void **ppFunction,
                           pModule **pThisModule
);
```

1.18.7 modu_UnloadAllModules

This function unloads all modules. This is called via the command finalizer mechanizm. If ever any module was loaded via a "declare sub" statement the command execution sets the command finalizer function pointer to point to this function.

```
int modu_UnloadAllModules(pExecuteObject pEo
);
```

In a multi-threaded environment this function calls the keeper function of the module and in case the keeper returns 1 the module is kept in memory, though the module finalizer function is called. This lets multi-thread external modules to keep themselfs in memory even those times when there is not any interpreter thread using the very module running.

In that case the module is put on the module list of the process SB object. That list is used to shut down the modules when the whole process is shut down.

If there is no process SB object (pEo->pEPo is NULL) then the variation is a single process single thread implementation of ScriptBasic. In this case this function first calls the module finalizer function that is usally called in multi-threaded environment every time an

interpreter thread is about to finish and after this the module shutdown function is called, which is called in a multi-thread environment when the whole process is to be shut down. After that the module is unloaded even if the keeper function said that the module wants to stay in memory.

Don't worry about this: it is not abuse. The keeper function saying 1 means that the module has to stay in memory after the actual interpreter thread has finished until the process finishes. However in this very case the process also terminates.

Note: A one-process one-thread implementation may also behave like a multi thread implementation allocating a separate process SB object and a program object to run. Then it should inherit the support table and the execution object of the process SB object to the runnable program object. After running finish the runned program object and call the shutdown process for the process SB object. But that is tricky for a single thread implementation.

1.18.8 modu_UnloadModule

This function unloads the named module. Note that this function is not called unless some extension module calls it to unload another module.

Currently there is no support for a module to unload itself.

```
int modu_UnloadModule(pExecuteObject pEo,
                      char *pszLibraryFile
);
```

1.18.9 modu_ShutdownModule

This function calls the shutdown function of a module.

If the shutdown function performs well and returns SUCCESS this function also returns success. If the shutdown function returns error code it means that the module has running thread and thus can not be unloaded.

```
int modu_ShutdownModule(pExecuteObject pEo,
                        pModule pThisModule
);
```

1.19 hookers.c

This file contains the hook functions that are called by the commands whenever a command wants to access the operating system functions. The hook functions implemented here are transparent, they call the operating system. However these hook functions are called via the HookFunctions function pointer table and external modules may alter this table supplying their own hook functions.

There are some hook functions, which do not exist by default. In this case the hook functions table points to `NULL`. These functions, if defined are called by ScriptBasic at certain points of execution. For example the function `HOOK_ExecBefore` is called each time before executing a command in case an external module defines the function altering the hook function table.

The hook functions have the same arguments as the original function preceeded by the pointer to the execution object `pExecuteObject pEo`. For example the function `fopen` has two arguments to `char *`, and therefore HOOK_fopen has three. The first should point to `pEo` and the second and third should point to

1.19.1 hook_Init

This function allocates a hook function table and fills the function pointers to point to the original transparent hook functions.

```
int hook_Init(pExecuteObject pEo,
              pHookFunctions *pHookers
);
```

1.19.2 hook_file_access

This function gets a file name as an argument and return an integer code that tells the caller if the program is allowed to read, write or both read and write to the file. The default implementation just dumbly answers that the program is allowed both read and write. This function is called by each other hook functions that access a file via the file name. If a module wants to restrict the basic code to access files based on the file name the module does not need to alter all hook functions that access files via file name.

The module has to write its own `file_access` hook function instead, alter the hook function table to point to the module's function and all file accessing functions will ask the module's hook function if the code may access the file.

The argument `pszFileName` is the name of the file that the ScriptBasic program want to do something. The actual `file_access` hook function should decide if the basic program is

0 not allowed to access the file

1 allowed to read the file

2 allowed to write the file (modify)

3 allowed to read and write the file

The default implementation of this function just allows the program to do anything. Any extension module may have its own implementation and restrict the basic program to certain files.

```
int hook_file_access(pExecuteObject pEo,
                     char *pszFileName
);
```

1.19.3 hook_fopen

```
FILE *hook_fopen(pExecuteObject pEo,
                 char *pszFileName,
                 char *pszOpenMode
);
```

1.19.4 hook_fclose

```
void hook_fclose(pExecuteObject pEo,
                 FILE *fp
);
```

1.19.5 hook_size

```
long hook_size(pExecuteObject pEo,
               char *pszFileName
);
```

1.19.6 hook_time_accessed

```
long hook_time_accessed(pExecuteObject pEo,
                        char *pszFileName
);
```

1.19.7 hook_time_modified

```
long hook_time_modified(pExecuteObject pEo,
                        char *pszFileName
);
```

1.19.8 hook_time_created

```
long hook_time_created(pExecuteObject pEo,
                        char *pszFileName
);
```

1.19.9 hook_isdir

```
int hook_isdir(pExecuteObject pEo,
               char *pszFileName
);
```

1.19.10 hook_isreg

```
int hook_isreg(pExecuteObject pEo,
               char *pszFileName
);
```

1.19.11 hook_fileexists

```
int hook_exists(pExecuteObject pEo,
                char *pszFileName
);
```

1.19.12 hook_truncate

```
int hook_truncate(pExecuteObject pEo,
                  FILE *fp,
                  long lNewFileSize
);
```

1.19.13 hook_fgetc

```
int hook_fgetc(pExecuteObject pEo,
               FILE *fp
);
```

1.19.14 hook_ferror

```
int hook_ferror(pExecuteObject pEo,
               FILE *fp
);
```

1.19.15 hook_fread

```
int hook_fread(pExecuteObject pEo,
               char *buf,
               int size,
               int count,
               FILE *fp
);
```

1.19.16 hook_setmode

```
void hook_setmode(pExecuteObject pEo,
                  FILE *fp,
                  int mode
);
```

1.19.17 hook_binmode

```
void hook_binmode(pExecuteObject pEo,
                  FILE *fp
);
```

1.19.18 hook_textmode

```
void hook_textmode(pExecuteObject pEo,
                   FILE *fp
);
```

1.19.19 hook_fwrite

```
int hook_fwrite(pExecuteObject pEo,
                char *buf,
                int size,
                int count,
                FILE *fp
);
```

1.19.20 hook_fputc

```
int hook_fputc(pExecuteObject pEo,
               int c,
               FILE *fp
);
```

1.19.21 hook_flock

```
int hook_flock(pExecuteObject pEo,
               FILE *fp,
               int iLockType
);
```

1.19.22 hook_lock

```
int hook_lock(pExecuteObject pEo,
              FILE *fp,
              int iLockType,
              long lStart,
              long lLength
);
  return file_lock(fp,iLockType,lStart,lLength);
```

1.19.23 hook_feof

```
int hook_feof(pExecuteObject pEo,
              FILE *fp
);
```

1.19.24 hook_mkdir

```
int hook_mkdir(pExecuteObject pEo,
               char *pszDirectoryName
);
```

1.19.25 hook_rmdir

```
int hook_rmdir(pExecuteObject pEo,
```

```
                char *pszDirectoryName
);
```

1.19.26 hook_remove

```
int hook_remove(pExecuteObject pEo,
                char *pszFileName
);
```

1.19.27 hook_deltree

```
int hook_deltree(pExecuteObject pEo,
                 char *pszDirectoryName
);
```

1.19.28 hook_MakeDirectory

```
int hook_MakeDirectory(pExecuteObject pEo,
                       char *pszDirectoryName
);
```

1.19.29 hook_opendir

```
DIR *hook_opendir(pExecuteObject pEo,
                  char *pszDirectoryName,
                  tDIR *pDirectory
);
```

1.19.30 hook_readdir

```
struct dirent *hook_readdir(pExecuteObject pEo,
                            DIR *pDirectory
);
```

1.19.31 hook_closedir

```
void hook_closedir(pExecuteObject pEo,
                   DIR *pDirectory
);
```

1.19.32 hook_sleep

```
void hook_sleep(pExecuteObject pEo,
                long lSeconds
);
```

1.19.33 hook_curdir

```
int hook_curdir(pExecuteObject pEo,
                char *Buffer,
                unsigned long cbBuffer
);
```

1.19.34 hook_chdir

```
int hook_chdir(pExecuteObject pEo,
               char *Buffer
);
```

1.19.35 hook_chown

```
int hook_chown(pExecuteObject pEo,
               char *pszFileName,
               char *pszOwner
);
```

1.19.36 hook_SetCreateTime

```
int hook_SetCreateTime(pExecuteObject pEo,
                       char *pszFileName,
                       long lTime
);
```

1.19.37 hook_SetModifyTime

```
int hook_SetModifyTime(pExecuteObject pEo,
                       char *pszFileName,
                       long lTime
);
```

1.19.38 hook_SetAccessTime

```
int hook_SetAccessTime(pExecuteObject pEo,
                       char *pszFileName,
                       long lTime
);
```

1.19.39 hook_gethostname

```
int hook_gethostname(pExecuteObject pEo,
                     char *pszBuffer,
                     long cbBuffer
);
```

1.19.40 hook_gethost

```
int hook_gethost(pExecuteObject pEo,
                 char *pszBuffer,
                 struct hostent *pHost
);
```

1.19.41 hook_tcpconnect

```
int hook_tcpconnect(pExecuteObject pEo,
                    SOCKET *sClient,
                    char *pszRemoteSocket
);
```

1.19.42 hook_tcpsend

```
int hook_tcpsend(pExecuteObject pEo,
                 SOCKET sClient,
                 char *pszBuffer,
                 long cbBuffer,
                 int iFlags
);
```

1.19.43 hook_tcprecv

```
int hook_tcprecv(pExecuteObject pEo,
                 SOCKET sClient,
                 char *pszBuffer,
                 long cbBuffer,
                 int iFlags
);
```

1.19.44 hook_tcpclose

```
int hook_tcpclose(pExecuteObject pEo,
                  SOCKET sClient
);
```

1.19.45 hook_killproc

```
int hook_killproc(pExecuteObject pEo,
                  long pid
);
```

1.19.46 hook_getowner

```
int hook_getowner(pExecuteObject pEo,
```

```
                    char *pszFileName,
                    char *pszOwnerBuffer,
                    long cbOwnerBuffer
);
```

1.19.47 hook_fcrypt

```
char *hook_fcrypt(pExecuteObject pEo,
                  char *buf,
                  char *salt,
                  char *buff
);
```

1.19.48 hook_CreateProcess

```
long hook_CreateProcess(pExecuteObject pEo,
                        char *pszCommandLine
);
```

1.19.49 hook_CreateProcessEx

```
long hook_CreateProcessEx(pExecuteObject pEo,
                          char *pszCommandLine,
                          long lTimeOut,
                          unsigned long *plPid,
                          unsigned long *plExitCode
);
```

1.19.50 hook_waitpid

```
int hook_waitpid(pExecuteObject pEo,
                 long pid,
                 unsigned long *plExitCode
);
```

1.19.51 hook_CallScribaFunction

This is a hook function that performs its operation itself without calling underlying `file_` function. This function is called by external modules whenever the external module wants to execute certain ScriptBasic function.

The external module has to know the entry point of the ScriptBasic function.

```
int hook_CallScribaFunction(pExecuteObject pEo,
                            unsigned long lStartNode,
                            pFixSizeMemoryObject *pArgument,
                            unsigned long NumberOfPassedArguments,
                            pFixSizeMemoryObject *pFunctionResult
);
```

1.20 options.c

Each BASIC interpreter maintains a symbol table holding option values. These option values can be set using the BASIC command `OPTION` and an option value can be retrieved using the function `OPTION()`.

An option has an integer value (`long`). Options are usually used to alter the behaviour of some commands or modules, altough BASIC programs are free to use any string to name an option. For example the option `compare` may alter the behavior of the string comparision function to be case sensitive or insensitive:

```
OPTION compare 1
```

Unitialized options are treated as being zero. There is no special option value for uninitialized options. In other words BASIC programs can not distinguish between unitialized options and options having the value zero.

This file contains the functions that handle the option symbol table. The option symbol tableis pointed by the field `OptionsTable` of the execution object. This pointer is initialized to be `NULL`, which means no options are available, or in other words all options are zero.

1.20.1 options_Reset

Calling this function resets an option. This means that the memory holding the `long` value is released and the pointer that was pointing to it is set `NULL`.

```
int options_Reset(pExecuteObject pEo,
                  char *name
);
```

1.20.2 options_Set

This function sets a long value for an option. If the option did not exist before in the symbol table it is inserted. If the symbol table was empty (aka `OptionsTable` pointed `NULL`) the symbol table is also created.

If the symbol already existed with some `long` value then the new value is stored in the already allocated place and thus the caller may store the pointer to the long returned by See ⟨undefined⟩ [GetR], page ⟨undefined⟩ and access possibly updated data without searching the table again and again.

```
int options_Set(pExecuteObject pEo,
                char *name,
                long value
);
```

The function returns zero if the option was set or `1` if there was a memory failure.

1.20.3 options_Get

This function retrieves and returns the value of an option data.

```
long options_Get(pExecuteObject pEo,
                 char *name
);
```

The return value is the option value or zero in case the option is not set.

1.20.4 options_GetR

This function retrieves and returns the value of an option data.

```
long *options_GetR(pExecuteObject pEo,
                   char *name
);
```

The return value is a `long *` pointer to the option value or `NULL` if the option is not set. If the caller sets the `long` variable pointed by the returned pointer the value of the option is changed directly.

1.21 report.c

This file contains a simple error report handling function that prints the error to the standard error.

This is a default reporting function used by most variations of ScriptBasic. However some variations like the ISAPI one needs to implements a function having the same interface.

1.21.1 report_report()

This function implements the default error reporting function for both run-time and parse time errors and warnings.

```
void report_report(void *filepointer,
                   char *FileName,
                   long LineNumber,
                   unsigned int iErrorCode,
                   int iErrorSeverity,
                   int *piErrorCounter,
                   char *szErrorString,
                   unsigned long *fFlags
);
```

Aguments:

`filepointer` is a `void *` pointer. The default value of this pointer is `stderr` unless the variation sets it different. This implementation uses this pointer as a `FILE *` pointer. Other implementations of this function may use it for any other purpose so long as long the usage of this pointer fits the variation.

`FileName` is the name of the source file where the error was detected. This parameter is `NULL` in case of a run-time error. The reporting function is encouraged to display this information for the user.

`LineNumber` is the line number within the source file where the error has happened. This parameter is valid only in case the parameter `FileName` is not `NULL`

`iErrorCode` is the error code.

`iErrorSeverity` should define the severity of the error. It can be `REPORT_INFO`, `REPORT_WARNING`, `REPORT_ERROR`, `REPORT_FATAL`, `REPORT_INTERNAL`. Whenever the error severity is above the warning level the `*piErrorCounter` has to be incremented.

`piErrorCounter` points to an `int` counter that counts the number of errors. If there are errors during syntax analysis the ScriptBasic interpreter stops its execution before starting execution.

`szErrorString` is an optional error parameter string and not the displayable error message. The error message is stored in the global constant array `en_error_messages`. This string may contain a `%s` control referring to the error parameter string.

`fFlags` is an `unsigned long` bit field. The bits currently used are: `REPORT_F_CGI` is set if the error is to be reported as a CGI script. See the code for more details. `REPORT_F_FRST` is reset when the report function is called first time and is set by the report function. This allows the report function to report a header in case it needs. Other bits are reserved for later use.

1.22 logger.c

This module can be used to log events. The module implements two type of logs.

synchronous logs

asynchronous logs

Synchronous logs are just the normal plain logging technic writing messages to a log file. This is low performance, because the caller has to wait until the logging is performed and written to a file. On the other hand this is a safe logging.

Asynchronous logging is a fast performance logging method. In this case the caller passes the log item to the logger. The logger puts the item on a queue and sends it to the log file in another thread when disk I/O bandwith permits. This is high performance, because the caller does not need to wait for the log item written to the disk. On the other hand this logging is not safe because the caller can not be sure that the log was written to the disk.

The program using this module should use asynchronous logging for high volume logs and synchronous logging for low volume logging. For example a panic log that reports configuration error has to be written synchronously.

Using this module you can initialize a log specifying the file where to write the log, send logs and you can tell the log to shut down. When shutting down all waiting logs are written to the file and no more log items are accepted. When all logs are written the logging thread terminates.

1.22.1 log_state()

This function safely returns the actual state of the log. This can be:

`LOGSTATE_NORMAL` the log is normal state accepting log items

`LOGSTATE_SHUTTING` the log is currently performing shut down, it does not accept any log item

`LOGSTATE_DEAD` the log is shut down all files are closed

`LOGSTATE_SYNCHRONOUS` the log is synchronous accepting log items

```
int log_state(ptLogger pLOG
);
```

1.22.2 log_init()

Initialize a log. The function sets the parameters of a logging thread. The parameters are the usual memory allocation and deallocation functions and the log file name format string. This format string can contain at most four `%d` as formatting element. This will be passed to `sprintf` with arguments as year, month, day and hour in this order. This will ease log rotating.

Note that log file name calculation is a CPU consuming process and therefore it is not performed for each log item. The log system recalculates the log file name and closes the old log file and opens a new one whenever the actual log to be written and the last log wrote is in a different time interval. The time interval is identified by the time stamp value divided (integer division) by the time span value. This is 3600 when you want to rotate the log hourly, 86400 if you want to rotate the log daily. Other rotations, like monthly do not work correctly.

To do this the caller has to set the `TimeSpan` field of the log structure. There is no support function to set this.

For example:

```
if( log_init(&ErrLog,alloc_Alloc,alloc_Free,pM_AppLog,CONFIG("log.err.file"),LOGT
  return 1;
if( cft_GetEx(&MyCONF,"log.err.span",&ConfNode,NULL,&(ErrLog.TimeSpan),NULL,NULL)
  ErrLog.TimeSpan = 0;
```

as you can see in the file `ad.c` Setting TimeSpan to zero results no log rotation.

Note that it is a good practice to set the TimeSpan value to positive (non zero) even if the log is not rotated. If you ever delete the log file while the logging application is running the log is not written anymore until the log file is reopened.

The log type can be `LOGTYPE_NORMAL` to perform asynchronous high performance logging and `LOGTYPE_SYNCHRONOUS` for syncronous, "panic" logging. Panic logging keeps the file continously opened until the log is shut down and does not perform log rotation.

```
int log_init(ptLogger pLOG,
             void *(*memory_allocating_function)(size_t, void *),
             void (*memory_releasing_function)(void *, void *),
             void *pMemorySegment,
             char *pszLogFileName,
             int iLogType
);
```

1.22.3 log_printf()

This function can be used to send a formatted log to the log file. The function creates the formatted string and then puts it onto the log queue. The log is actually sent to the log file by the asynchronous logger thread.

```
int log_printf(ptLogger pLOG,
               char *pszFormat,
               ...
);
```

1.22.4 log_shutdown()

Calling this function starts the shutdown of a log queue. This function allways return 0 as success. When the function returns the log queue does not accept more log items, however the queue is not completely shut down. If the caller wants to wait for the queue to shut down it has to wait and call the function See ⟨undefined⟩ [log_state], page ⟨undefined⟩ to ensure that the shutdown procedure has been finished.

```
int log_shutdown(ptLogger pLOG
);
```

1.23 thread.c

This file implements global thread handling functions. If the programmer uses these functions instead of the operating system provided functions the result will be Windows NT *and* UNIX portable program. These routines handling thread and mutex locking functions had been extensively tested in commercial projects.

1.23.1 thread_CreateThread

This is a simplified implementation of the create thread interface.

The function creates a new **detached** thread. If the thread can not be created for some reason the return value is the error code returned by the system call `pthread_start` on UNIX or `GetLastError` on NT.

If the thread was started the return value is 0.

```
int thread_CreateThread(PTHREADHANDLE pThread,
                        void *pStartFunction,
                        void *pThreadParameter
);
```

The arguments

`pThread` is a thread handle. This should be a pointer to a variable of type `THREADHANDLE`. This argument is set to hold the thread handle returned by `CreateThread` on NT or the pointer returned as first argument of `pthread_create` under UNIX. This argument is not used further in this module but can be used if calling system dependant functions.

`pStartFunction` should be a pointer pointing to the start function where the thread should start. This is usually just the name of the function to start in the separate thread.

`pThreadParameter` is the pointer passed as argument to the start function.

1.23.2 thread_ExitThread

Exit from a thread created by See ⟨undefined⟩ [CreateThread], page ⟨undefined⟩. The implementation is simple and does not allow any return value from the thread.

```
void thread_ExitThread(
);
```

1.23.3 thread_InitMutex

This function initializes a `MUTEX` variable. A `MUTEX` variable can be used for exclusive access. If a mutex is locked another lock on that mutex will wait until the first lock is removed. If there are several threads waiting for a mutex to be released a random thread will get the lock when the actually locking thread releases the mutex. In other words if there are several threads waiting for a mutex there is no guaranteed order of the threads getting the mutex lock.

Before the first use of a `MUTEX` variable it has to be initialized calling this function.

```
void thread_InitMutex(PMUTEX pMutex
);
```

Arguments:

`pMutex` should point to a mutex variable of the type `MUTEX`

1.23.4 thread_FinishMutex

When a mutex is not used anymore by a program it has to be released to free the system resources allocated to handle the mutex.

```
void thread_FinishMutex(PMUTEX pMutex
);
```

Arguments:

`pMutex` should point to an initialized mutex variable of the type `MUTEX`

1.23.5 thread_LockMutex

Calling this function locks the mutex pointed by the argument. If the mutex is currently locked the calling thread will wait until the mutex becomes available.

```
void thread_LockMutex(PMUTEX pMutex
);
```

Arguments:

`pMutex` should point to an initialized mutex variable of the type `MUTEX`

1.23.6 thread_UnlockMutex

Calling this function unlocks the mutex pointed by the argument. Calling this function on a mutex currently not locked is a programming error and results undefined result. Different operating system may repond different.

```
void thread_UnlockMutex(PMUTEX pMutex
);
```

Arguments:

`pMutex` should point to an initialized mutex variable of the type `MUTEX`

1.23.7 thread_shlckstry

The following functions implement shared locking. These functions do not call system dependant functions. These are built on the top of the MUTEX locking functions.

A shareable lock can be **READ** locked and **WRITE** locked. When a shareable lock is READ locked another thread can also read lock the lock.

On the other hand a write lock is exclusive. A write lock can appear when there is no read lock on a shareable lock and not write lock either.

The story to understand the workings:

Imagine a reading room with several books. You can get into the room through a small entrance room, which is dark. To get in you have to switch on the light. The reading room has a light and a switch as well. You are not expected to read in the dark. The reading room is very large with several shelves that easily hide the absent minded readers and therefore the readers can not easily decide upon leaving if they are the last or not. This actually led locking up late readers in the dark or the opposite: lights running all the night.

To avoid this situation the library placed a box in the entrance room where each reader entering the room have to place his reader Id card. When they leave they remove the card. The first reader coming switches the light on, and the last one switches the light off. Coming first and leaving last is easily determined looking at the box after dropping the card or after taking the card out. If there is a single card after dropping the reader card into you are the first coming and if there is no card in it you took your one then you are the last.

To avoid quarreling and to save up energy the readers must switch on the light of the entrance room when they come into and should switch it off when they leave. However they have to do it only when they go into the reading room, but not when leaving. When someone wants to switch a light on, but the light is already on he or she should wait until the light is switched off. (Yes, this is a MUTEX.)

When the librarian comes to maintain ensures that no one is inside, switches the light of the entrance room on, and then switches the reading room light on. If someone is still there he cannot switch the light on as it is already switched on. He waits until the light is switched off then he switches it on. When he has switched the light of the reading room on he switches the light of the entrance room off and does his job in the reading room. Upon leaving he switches off the light of the reading room.

Readers can easily enter through the narrow entrance room one after the other. They can also easily leave. When the librarian comes he can not enter until all readers leave the

reading room. Before getting into the entrance room he has equal chance as any of the readers.

1.23.8 thread_InitLock

```
void shared_InitLock(PSHAREDLOCK p
);
```

1.23.9 thread_FinishLock

```
void shared_FinishLock(PSHAREDLOCK p
);
```

1.23.10 thread_LockRead

```
void shared_LockRead(PSHAREDLOCK p
);
```

1.23.11 thread_LockWrite

```
void shared_LockWrite(PSHAREDLOCK p
);
```

1.23.12 thread_UnlockRead

```
void shared_UnlockRead(PSHAREDLOCK p
);
```

1.23.13 thread_UnlockWrite

```
void shared_UnlockWrite(PSHAREDLOCK p
);
```

1.24 hndlptr.c

The functions in this file help the various ScriptBasic extension modules to avoid crashing the system even if the BASIC programs use the values passed by the module in a bad way.

For example a database handling module opens a database and allocates a structure describing the connection. The usual way to identify the structure is to return a BASIC string variable to the BASIC code that byte by byte holds the value of the pointer. This works on any machine having 32bit or 64bit pointers because strings can be arbitrary length in ScriptBasic.

When another external module function need access to the structure it needs a pointer to it. This is easily done by passing the string variable to the module. The module converts the string variable back byte by byte to a pointer and all is fine.

Is it?

The issue is that the BASIC program may alter the pointer and pass a string containg garbage back to the module. The module has no way to check the correctness tries to use it and crashes the whole interpreter. (Even the other interpreters running in the same process in different threads.)

=bold ScriptBasic external modules should never ever pass pointers in strings back to the BASIC code. =nobold

(Even that some of the modules written by the ScriptBasic developers followed this method formerly.)

The better solution is to store these module pointers in arrays and pass the index of the pointer in the array to the basic application. This way the BASIC program will get INTEGER values instead of STRING and will not be able to alter the pointer value and crash the program.

To store the pointer and get the index (we call it a handle) these functions can be used.

Whenever a pointer needs a handle the module has to call `GetHandle`. This function stores the pointer and returns the handle to it. When the BASIC program passes the handle back to the module and the module needs the pointer associated with the handle it has to call `GetPointer`.

When a pointer is not needed anymore the handle should be freed calling `FreeHandle`.

This implementation uses arrays to hold the pointers. The handles are the indexes to the array. The index 0 is never used. Handle value zero is returned as an invalid handle value whenever some error occures, like out of memory condition.

1.24.1 handle_GetHandle

Having a pointer allocate a handle. This function stores the pointer and returns the handle.

The handle is a small positive integer.

If any error is happened (aka out of memory) zero is returned.

```
unsigned long handle_GetHandle(void **pHandle,
                               void *pMEM,
                               void *pointer
);
```

The first argument `pHandle` is a pointer to the handle array.

The second argument `pMEM` is the memory segment that is to be used to allocate memory.

The last argument `pointer` is the pointer to store.

Note that `NULL` pointer can not be stored in the array.

The pointer to the handle array `pHandle` should be initialized to NULL before the first call to `handle_GetHandle`. For example:

```
void *Handle = NULL;
  ....
if( !handle_GetHandle(&Handle,pMEM,pointer) )return ERROR_CODE;
```

1.24.2 handle_GetPointer

This function is the opposite of See ⟨undefined⟩ [GetHandle], page ⟨undefined⟩. If a pointer was stored in the handle array this function can be used to retrieve the pointer knowing the handle.

```
void *handle_GetPointer(void **pHandle,
                        unsigned long handle
);
```

The first argument `pHandle` is the pointer to the handle array. =ite, The second argument `handle` is the handle of the pointer.

If there was not pointer registered with that handle the return value of the function is `NULL`.

1.24.3 handle_FreeHandle

Use this function when a pointer is no longer valid. Calling this function releases the `handle` for further pointers.

```
void handle_FreeHandle(void **pHandle,
                       unsigned long handle
);
```

1.24.4 handle_DestroyHandleArray

Call this function to release the handle array after all handles are freed and there is no need for the handle heap.

Use the same memory head `pMEM` that was used in See ⟨undefined⟩ [GetHandle], page ⟨undefined⟩.

```
void handle_DestroyHandleArray(void **pHandle,
                               void *pMEM
);
```

1.25 httpd.c

1.25.1 httpd module

This module is used only by the standalone webserver variation of ScriptBasic.

The module contains a function See ⟨undefined⟩ [httpd], page ⟨undefined⟩ that the main application should start. This function calls the initialization function See ⟨undefined⟩ [AppInit], page ⟨undefined⟩ and the application starting functios See ⟨undefined⟩ [AppStart], page ⟨undefined⟩. After See ⟨undefined⟩ [AppStart], page ⟨undefined⟩ returns it starts to listen on the configured port and accepts http requests and passes the requests to See ⟨undefined⟩ [HttpProc], page ⟨undefined⟩.

1.25.2 AppInit

=bold This function is not implemented in this module. This function is used by this module and the program using this module should provide this function. =nobold

This function is called by the function See ⟨undefined⟩ [httpd], page ⟨undefined⟩ practically before anything is done.

```
int AppInit(int argc,char *argv[],pHttpdThread pHT,void **AppData),
```

The See ⟨undefined⟩ [httpd], page ⟨undefined⟩ function passes the command line arguments back as it gets them plain. The pointer `pApp` points to an applicatrion specific `void` pointer that is initialized to be `NULL` and is guaranteed not been changed. The pointer to the same `void` pointer is passed also to See ⟨undefined⟩ [AppStart], page ⟨undefined⟩. This pointer should be used to pass data between `AppInit` and See ⟨undefined⟩ [AppStart], page ⟨undefined⟩.

The pointer `pHT` points to a `HttpThread` structure and the function `AppInit` can change the values of this structure.

The entry point of the function `AppInit` should be given to the function See ⟨undefined⟩ [httpd], page ⟨undefined⟩ as argument.

1.25.3 AppStart

=bold This function is not implemented in this module. This function is used by this module and the program using this module should provide this function. =nobold

This function is called by See ⟨undefined⟩ [httpd], page ⟨undefined⟩ after binding on the desired port, and after forking to background on UNIX. This function should start all threads instead of See ⟨undefined⟩ [AppInit], page ⟨undefined⟩, otherwise the forking looses all threads except the main thread. The first version of this code started the logger threads before the fork and the parent exited with the running logger threads whil the child daemon did not run the logger threads.

```
int AppStart(void **pApp);
```

1.25.4 HttpProc

=bold This function is not implemented in this module. This function is used by this module and the program using this module should provide this function. =nobold

This function is called by See ⟨undefined⟩ [httpd], page ⟨undefined⟩ for each hit in a separate thread.

```
void HttpProc(pHttpdThread pHT,pThreadData ThisThread);
```

1.25.5 FtpProc

=bold This function is not implemented in this module. This function is used by this module and the program using this module should provide this function. =nobold

This function is called by See ⟨undefined⟩ [httpd], page ⟨undefined⟩ for each ftp command.

```
void FtProc(pHttpdThread pHT,pThreadData ThisThread, char *pszCommand);
```