ScriptBasic Users Guide

Peter Verhas

Short Contents

Table of Contents

1 Introduction

ScriptBasic is a BASIC language interpreter with several features that makes it unique. First of all ScriptBasic itself is **free** and is supported by the **GNU LGPL licence**. Note that the GNU LGPL licence (`http://www.gnu.org/copyleft/lesser.txt`) applies only to ScriptBasic itself, while the modules interfacing 3rd party software may apply different licenses. For example the module interface code for the module T<bdb> is under LGPL, but the library it uses, namely the Berkeley DB is NOT LGPL!

ScriptBasic runs on **Windows NT, Windows95, Linux, Tru64 UNIX** and probably on many other platforms. ScriptBasic the ideal tool to write small scripts that gurus write in Perl. However you need not be a guru to powerfully program in ScriptBasic.

This is not the only situation to consider ScriptBasic. ScriptBasic can be a valuable tool for experts as well as a language interpreter ready to be built into their application. Read the list of ScriptBasic features and decide how you can use it.

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

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

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

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

COMPILED CODE ScriptBasic creates intermediate compiled code, which is interpreted afterwards. This can protect intellectual property for the BASIC programmer and faster code loading.

Syntax analysis is done at first and only syntactically perfect programs start to run.

The compiled code is put into a continuous memory space and compiled code can be saved and loaded again to run without recompilation. This is vital for CGI scripts and is not available for most scripting programming languages.

Compiled code is binary, not readable. Therefore you can develop and distribute programs and getting some help to protect your intellectual property. You need not give the source code.

MULTI THREAD aware. Although the current implementation is not multi thread, all the code was designed to be thread safe. You can embed the code into systems that run multiple interpreters in the same process. On the other hand the interpreter

can run the same code in multiple threads and was designed to be capable handling call-back functions, and multithread programs in the future.

This documentation is the User's Guide for the so-called STANDARD variation of the interpreter. This code runs on the command line, accepts command line arguments and runs a single program in a single process. Other variations exist, which are based on the same code but exhibit different interfaces to the system. Some features for those variations may be different, but most language features probably remain the same. The variation Eszter SB Application Engine is shipped with the ScriptBasic package and is embedding the interpreter into a multi-thread, single process web server. Other embedded variations are available from independent developers.

This document describes how to use the interpreter and the programming language.

2 Using ScriptBasic

This chapter is about the STANDARD variation of the interpreter. Starting and using other variations may probably be different.

2.1 Running from command line

There are several ways to start a ScriptBasic program. The different ways also depend on the installation. Windows NT may associate the extension T<.sb> with the ScriptBasic interpreter. In this case you can start a ScriptBasic program double clicking on its name in the explorer. You can type the name of the text file containing your basic program and UNIX will start it for you automatically if the first line of the program contains the starting information like

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

and the file itself is executable. The simplest way among all is to type the name of the executable. First try to start it without any command line argument:

```
$ scriba
Usage: basic [options] program.bas

options: -o file_name
            specify output file, save binary format to file but don't execute
         -b file_name
            load binary format from file and execute
         -n
            do not use cache (no save, no load)
         -e
            execute after binary format was saved
         -v
            print version info and stop
         -c
            inform scriba that this is a CGI script.
         -C
            save C program output.
         -E
            save executable output. (may not work under some OS)
         -p preprocessor
            specify external preprocessor.
         -i preprocessor
            specify internal preprocessor.
         -f configurationfile
            specify configuration file
         -d
            debug module error (UNIX only)
         -k text_config_file
            compile the configuration file to binary
```

```
-D
   dump the configuration file in text format
```

The program does not insist on its name, the person installing can name it `basic`, `scriba` or any other name. However `scriba` is the preferred name for the executable. When you start the program without arguments it tells you the different options. If the program does not start check that the executable is in the path or specify the full path to the executable. Under UNIX you may need to start ScriptBasic after you have compiled as `./scriba`

The compilation and execution of the code can be altered and driven by command line options. There are not too many and usually you do not need any. If you want to start a basic program you can type:

```
$ scriba hello.bas
Hello world!
$ _
```

The program will execute the basic code and after executing the final statement it exists the process.

OPTION -o

ScriptBasic STANDARD variation is able to dump the compiled code to a file and later use this image to execute the basic program. To do this you can type

```
$ scriba -o hello.bbf hello.bas
```

This command will compile the program `hello.bas` into the file `hello.bbf`. The code itself will not be executed. The extension `bbf` stands for *basic binary format*, but you can use any extension. ScriptBasic does not assume any default extensions.

OPTION -b

The saved binary format can be executed by the interpreter issuing the command

```
$ scriba -b hello.bbf
```

The switch '`-b`' tells the interpreter that the file is already a compiled format and it needs only execution. However ScriptBasic STANDARD variation is intelligent enough to recognize the binary format and it will not try to interpret it as basic text even if you miss the switch '`-b`'. The command line

```
$ scriba hello.bbf
```

should also work. The switch '`-b`' is available because recognizing the binary format applies some heuristics and using the '`-b`' switch to execute a compiled basic program is the safe way.

Note that the compiled format is not compiled in the sense of usual compilers. The compiled format is NOT machine executable. This is the internal format that the interpreter interprets and executes. This format is created and stored in memory each time a basic program is executed. When the '`-o`' switch is used this internal format is saved and can be reloaded later to execute again the basic program.

This binary format is proprietary and is NOT portable. This binary format was designed to load the programs fast and allow the interpreter speedy startup for codes executed many times. You have to "compile" your basic code on the architecture you want to run it. It needs not be physically the same machine, but it should be the same type of operating system, the same compiler used to compile ScriptBasic executable and the same ScriptBasic

version and build number. The different variations may accept each others binary format. For example the Eszter SB Application Engine variation accepts the binary format created by the STANDARD variation. We recognized that the STANDARD variation running on Linux compiled with `gcc` can execute the binary format generated by the Windows NT version of ScriptBasic, but this is only the fact of life and it is not a guaranteed feature.

OPTION -e

In certain situations you may want to save the binary format and execute the code the same time. In this situation you can issue the command line:

```
$ scriba -o hello.bbf -e hello.bas
```

The switch '`-e`' tells ScriptBasic to execute the code. This is **not** the normal behavior when binary format is saved to file. Note that you need the switch '`-e`' only when the switch '`-o`' is also on the command line. To execute a program without saving the binary format to a file you need no switches at all.

OPTION -n

The binary format of the code is usually saved into a directory without user intervention. The cache directory is defined in the configuration file using the configuration key `cache`. In certain situation you want to avoid caching of the code. If you use the command line option '`-n`' neither cache checking for newer version nor cache writing will be performed.

OPTION -c

You should use the '`-c`' option in programs that are executed as CGI scripts. This helps the error reporting system to cooperate with the web server and send properly formatted HTML formatted errors whenever and error occurs.

OPTION -C

The option '`-C`' asks the interpreter to generate C language compiled code. For more about this option read in See ⟨undefined⟩ [Compiling BASIC programs to C], page ⟨undefined⟩.

OPTION -E

The option '`-E`' asks the interpreter to create a stand alone executable from the BASIC program. This is an option that works only on some platform, but not on all. It does work on Windows and under Linux and some users reported success using it under FreeBSD and Solaris. When using this option the interpreter creates a file that can be started and contains the compiled BASIC program in itself. This is not a compilation just as well as compiling into binary format is not a real compilation. It copies the binary executable code of the interpreter and the binary format of the BASIC program into one file and sets it so that executes the BASIC program embedded into the executable file.

When you use this option under Linux, you have to specify the full path to the interpreter. Thus you may not successfully execute

```
scriba Eo test test.sb
```

but rather you have to type:

```
/usr/bin/scriba Eo test test.sb
```

This is because ScriptBasic relies on the command line to find the executable file it has to copy into the created output executable. If you do not know where the ScriptBasic interpreter is installed you can type the command:

```
which scriba
```

to find out the exact location.

Also note that the generated file can not be executed immediately after it has been generated. You have to set the execute permission on the generated file, for example:

```
chmod u+x test
```

before you can issue the command

```
./test
```

OPTION -p

The option '`-p`' should be used to specify an external preprocessor. This preprocessor has to be configured in the configuration file of ScriptBasic. Ask the system manager responsible for your installation about the available preprocessors. If this person is you read the section See ⟨undefined⟩ [Using external preprocessor], page ⟨undefined⟩. You can specify more than one preprocessors on the line. These are executed one after the other in the order they are specified on the command line. If no preprocessor is defined on the command line the interpreter will execute the preprocessor or preprocessors that are assigned to the extensions of the source file name. The assignments should be configured in the ScriptBasic configuration file.

OPTION -i

The option '`-i`' specifies an internal preprocessor. Internal pre-processors are implemented as DLL or SO file. These programs are loaded by the ScriptBasic interpreter when the command line option '`-i`' is used, or when the program contains the command `USE`. The internal pre-processor can alter the source code after the interpreter read the code into memory, but also can do a lot of other things while ScriptBasic compiles the code, and some of them even during run time. For more information on internal pre-processors read the guide of the actual pre-processor, or read the developers guide that details how internal pre-processors should be developed.

The argument to this option has to be the name of the internal preprocessor as configured in the configuration file. For example, to start the program in command line debugger mode the user has to type:

```
scriba -i dbg hello.bas
```

The program will load the preprocessors in the order they are specified.

OPTION -f

When ScriptBasic starts it reads its configuration file. Some programs can run without any configuration information, however more complex programs will need external modules and this most probably will require configuration. The configuration file is located usually at '`/etc/scriba/basic.conf`' under UNIX or in the same directory where the executable is under Windows NT. However in some situation you may want to run ScriptBasic using some special configuration file. In this case you can specify the configuration file using the option '`-f`'. The argument to this option is the name of the file that is the compiled configuration information. The only exception is when you are using this option along with the option '`-k`' to compile the configuration information. In this case the argument to the option '`-f`' has to be the name of the configuration file that is to be created.

OPTION -d

The option '`-d`' is a special one and you need it only when you try to find out configuration errors you made. The option can be used under UNIX as well as under Windows. However ScriptBasic will print out different errors under the different operating systems.

You can also use this option when a configuration file can not be compiled and ScriptBasic reports that there are errors in the configuration file. Be prepared that ScriptBasic may print out quite a huge portion of the configuration file following the error to specify the location of the error.

On UNIX the option '`-d`' prints out the system error message when an external module fails to load. Without this option the interpreter says: 'module can not be loaded', but you, the installer of the module do not know why. Here is the reason why it is needed in some cases on UNIX and why not under Windows.

On Windows when you compile source code to DLL all symbolic references have to be resolved. If a module uses the `zlib.lib` for example, you have to link it against the DLL during link time. UNIX is more allowing. If you miss the '`-lz`' (to stay with the example of needing `zlib`) when you link the shared object the linker does not complain, because UNIX dynamic loading system allows symbolic reference resolution during run time. However when you run ScriptBasic and want to load the module this missing function appears immediately.

A real word example is the MySQL module. The first version of ScriptBasic MySQL module supported MySQL v3.22. The version v3.23 already required the `zlib` library. The v2.23 was tested only on Windows NT and when it was first compiled by a FreeBSD user, the module was not loadable. He needed the '`-d`' to see the error, which said 'unresolved reference: uncompress' making immediately clear that the option '`-lz`' had to be added to the `UXLIBS:` line in the file `interface.c`.

Using the option '`-d`' ScriptBasic will print out the directories where it seeks include files and modules.

OPTION -k

The option '`-k`' can and should be used to compile a text configuration file to binary format. In this case the input file name should specify a text format configuration file and not a BASIC program. The output file can be specified by the option '`-f`', but it is not a must. If the option '`-f`' is not used along with the option '`-k`' the output file will be the default configuration file.

Note that ScriptBasic command line parsing is the same when you use the option '`-k`' as it is when executing a program. Therefore all options, typically the option '`-f`' should precede the input file name. If the option '`-f`' is placed after the input file name, the command line parsing program believes that it will be some command line argument for the BASIC program and ignores it.

OPTION -D

The option '`-D`' can be used to display the content of the current compiled configuration file. ScriptBasic will read the configuration file and dump the content formatted according to the text configuration syntax to the standard output. You can use the option '`-f`' to specify an alternate compiled configuration file if not the default is to be debugged. If an output file is specified using the option '`-o`' along with the option '`-D`' the text content of the configuration file is printed into the output file. The output can be sent to the standard

output explicitly specifying `-o STDOUT` as output or to the standard error specifying `-o STDERR`.

OPTION -v

The last switch is '`-v`', which can be used to print version information.

```
$ scriba -v


ScriptBasic v2.0
Variation >>STANDARD<< build 1
Magic value 859010868


Node size is 16
Extension interface version is 11
Compilation: Jul 23 2002 22:24:56
Executable: T:\ScriptBasic\bin\scriba.exe
```

When you use this switch ScriptBasic does not execute any program. Instead it prints out information on itself and stops. The information lines printed are:

`ScriptBasic v2.0` It is ScriptBasic version 2.0

`Variation >>STANDARD<< build 1` The variation is STANDARD and this code is based on the build 1 source version. The build number is a sub version. Therefore you may think of it as this is version v2.0.1

Note that the actual version and build may and most probably is different from the example presented here. This is a printout of the version by the time this part of the documentation was written.

`Magic value 859010868` This is a magic code calculated from the date of the compilation of ScriptBasic. Currently it is not used.

`Node size is 16` The size of a node in bytes of the internal format. When a basic program is compiled and saved using the option '`-o`' several "nodes" are saved. On different platforms the size of the node is different. This is actually the size of the C `struct` defined in '`build.c`'

This is usually 16bytes on 32 bit machines. If you get a higher value on a 32bit-machine when you compile ScriptBasic from source check your compiler for alignment options.

Tru64 UNIX compiler creates 24bytes nodes.

If you do not understand all this node size stuff, just forget about it.

`Extension interface version is 11` The interface version that ScriptBasic currently uses to load external modules. The external modules are dynamic libraries that can be loaded during run time. On UNIX they usually have the extension '`.so`'. On Windows they have the extension '`.dll`'. The interface version should not be confused with the version of ScriptBasic or the version of a module. This is the version of the interface that ScriptBasic and the module should agree.

`Compilation: Jul 23 2002 22:24:56` The time when the executable was compiled CET.

The actual magic value and the compilation date will be different when you try the option '`-v`' on the released version of ScriptBasic, because this documentation was created using a pre-release version of ScriptBasic v2.0b1.

Note that any command line option should be specified before the name of the BASIC program. Any command line option or argument specified following the name of the file containing the basic program in text or binary format will be passed to the basic program itself. The basic program can use the function `COMMAND()` to access this part of the command line.

2.2 Running CGI programs

Although this topic should have been out of the scope of this documentation it is described here in detail because of its importance. Several users want to run ScriptBasic programs as CGI scripts.

Running CGI programs is just the same as running command line programs if you are experienced configuring your web server. The major difference is that it is the web server that starts the command line program and not the shell (or command.exe).

Under UNIX you can start a ScriptBasic program the same way as any other CGI scripts. The very first line of BASIC program should be

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

assuming that the executable code is called scriba and it is placed in the directory /usr/bin. The text file containing the code should also be executable for the user who runs the CGI scripts. This user is usually called nobody. You may say

```
$ chmod a+x hello.bas
```

to give all users execute permission. This may impose security questions that we do not discuss here. You should really know what you are doing.

On Windows NT using Internet Information Server the situation is different. Here you have to associate the extension with the executable of ScriptBasic. Note that this is not the same association as the one that allows the explorer to run the program when you double click on it. You have to configure the association in the Internet Service Manager configuration program.

You can also execute BASIC programs in the Eszter SB Application Engine to generate web applications. In this case the Eszter SB Application Engine has to be started from the command line or as a daemon under UNIX or installed as a service under Windows NT. The BASIC programs are executed inside the engine without starting a new process unlike in CGI and thus execution is faster. Nevertheless the programs on the BASIC programming level feel as if they were CGI programs, thus CGI programs can be executed this way without any modification.

2.3 Writing CGI programs

There are two ways to write CGI programs in ScriptBasic. The old way is just like in any other language: the basic program can access the environment, the standard input and

standard output. This is all needed to write a CGI program, decode the CGI parameters and create the http response.

The better way is to use the CGI module delivered with ScriptBasic that automatically handles CGI input, CGI environment variables, creates the uploaded files and even supports some security settings. For more details on the CGI module read the separate documentation of the module named CGI.

3 Installation Instructions

Installation of ScriptBasic is easy. Although there is no `SETUP.EXE` to install the program, it does not require you to be rocket scientists to install under Windows NT. Under Linux you can use the Debian or the RedHat packages to install the program. Under other unices you should build the and install the program from source that should be as complex and sophisticated as typing make install.

3.1 Installation under Windows

To install the software on Windows is quite easy. Extract the ZIP file into a temporary directory and start the executable '`SETUP.EXE`'.

Note that another file named '`SBCAB.BIN`' should be in the same directory where the program '`SETUP.EXE`' is. This file '`SBCAB.BIN`' contains the files in a compressed archive that the program '`SETUP.EXE`' reads and uses to install ScriptBasic.

After starting '`SETUP.EXE`' all you have to do is to specify the directory where you want to install ScriptBasic, and '`SETUP.EXE`' does the rest for you.

After you have installed ScritpBasic you can safely delete the files '`SETUP.EXE`' and '`SBCAB.BIN`' from the temporary directory. Do not delete '`SETUP.EXE`' that was copied into the installation directory. This is needed to uninstall ScriptBasic in case you want to uninstall it ever.

3.1.1 Installation under Windows from source

To install from sources you have to compile ScriptBasic first. The process of compilation under Windows operating system is defined in the Developers Guide. Note that the Developers Guide may not be up to date lacking this chapter.

ScriptBasic does not require installation to use the basic features. Installation and configuration is required to enable ScriptBasic to use advanced features like, loading external modules.

After compilation you can run the BASIC program '`install.sb`' using the freshly compiled executable '`scriba.exe`'. This program will partially install ScriptBasic into a directory. To start the program use the command file '`install.cmd`'.

After you have done this you may want to alter the system `PATH` to include the directory where the executable is. In case you use the default installation directory this is

```
C:\ScriptBasic\bin
```

On Windows 2000 to do it click on the start button, control panel, system. Choose the tab Advanced and then the middle button with the text `Environment Variables...`

Select the variable `Path` either from User variables of from System variables. Press the button `Edit...`, press `"Home"` to get to the start of the line and type the directory of the ScriptBasic binaries, like `C:\ScriptBasic\bin;`. Do not delete any of the characters already in the edit box, and do not forget the separating semicolon.

You can find a file '`scriba.conf`' in this directory. This is the configuration file of ScriptBasic. Do NOT try to edit this file using notepad. This is a binary format file created

using the ScriptBasic command line option '`-k`'. If you want to change any configuration options edit the file '`scriba.conf.lsp`' and generate the new '`scriba.conf`' file typing the command line:

```
C:\ScriptBasic\bin> scriba -k scriba.conf.lsp
```

When ScriptBasic starts it searches for the configuration file in the same directory where the executable is (this is a Windows only feature). If you want to store the configuration file in a different directory then use the registry editor and set the key

```
HKEY_LOCAL_MACHINE\Software\ScriptBasic\config
```

to hold the full path file name of the configuration file (not the text version but the converted binary).

After the successful installation run some test programs.

3.2 Installation under Linux

To install ScriptBasic under Debian Linux is as simple as typing

```
dpkg --install scriba-v2.0b1-1_i386.deb
```

If you happen to install ScriptBasic on a RedHat Linux then you can download the RPM version of the program and say

```
rpm -i scriba-2.0b1-1.i386.rpm
```

This will install and configure ScriptBasic form most of the systems. In case the directory structure of the standard installation does not fit your needs then you can edit the configuration file scriba.conf.unix.lsp and convert it to binary version using the command line:

```
#/usr/bin/scriba -k scriba.conf.unix.lsp
```

This is the default configuration file that ScriptBasic reads and all other directories and locations are taken from this file.

In case you want to create a version of ScriptBasic that links some external modules statically or want to play around with the source then you have to recompile the source. Because this is the standard installation mode for any operating system that is not Windows nor Linux, read on in the next section how to do it.

3.3 Installation under other Unices

On Windows NT and Linux (Debian or RedHat) the installation is simple. On other operating system the installation is a bit more sophisticated. You have to extract the `tar.gz` source file into a directory get super-user privileges issuing the command `su` and have to type:

```
./setup
```

If you happen to need it try it now. If you face issues, then please consult the developers guide that details how to compile ScriptBasic sources.

3.4 Configuration

To reach advanced features, like using system include files from predefined directories or using dynamic modules you have to create a configuration file. This configuration file is probably different for most installation and reflects the directory structure, networking environment and other specialties of your setup.

The format of the configuration file is binary. This is to speed up configuration information reading. To create this binary format, you have to edit the text format of the configuration file usually named '`scriba.conf.lsp`' and have to convert it to binary format using the command line

```
scriba -k scriba.conf.lsp
```

The ScriptBasic executable using the option '`-D`' dumps binary configuration files into text format. Note that this dump program will not restore the comments of the original configuration file nor the line breaks or spaces (except those in strings).

The format of the text file version of the configuration information is simple. It contains the keys and the corresponding values separated by one or more spaces and new lines. Usually a key and the assigned value are written on a line. Lines starting with the character `;` are comment.

The values can be integer numbers, real numbers, strings and sub-configurations. Strings can either be single line started and terminated by a single `"` character or multi-line strings starting and ending with three `"""` characters, just like in the language ScriptBasic.

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

[Former versions of ScriptBasic preceding version 2.0.0 did not allow a configuration to have empty sub configuration. Version 2.0.0 and later allow such a configuration part. This may happen in case the configuration file is edited and all the keys in a subconfiguration is commented out. This was a bug that was hard to spot. The later version executed a clean up procedure when reading the text version of the configuration file during configuration compilation and removes the sub tree.]

The keywords of the current version are:

`dll` defines the extension used for dynamic load libraries. This extension is used when a module is loaded and full path file name is not specified. If there are more than one extension specified only the first one is used.

The string defined in this key is appended to the file name when ScriptBasic is loading a module. ScriptBasic does NOT insert the dot before the extension; therefore the key value should contain it. Thus the correct configuration is

```
dll ".dll"
```

`module` defines the directory for dynamically loadable modules. You should define the full path to the dynamic load libraries. In the path name you can use forward or back slashes.

ScriptBasic does not insert a slash between the file name and the directory path, therefore the key value should contain the trailing slash.

If there are more than one values defined for this key ScriptBasic will search for the module in each directory defined in the order they are defined. The search finishes when the module is found and successfully loaded or if there are no more directories.

`include` defines the directory for system or module-include files. These files are included in the include statement specifying the file name as bare word without double quotes. You should define the full path to the files. In the path name you can use forward or back slashes.

ScriptBasic does not insert a slash between the file name and the directory path, therefore the key value should contain the trailing slash.

If there are more than one values defined for this key ScriptBasic will search for the module in each directory defined in the order they are defined. The search finishes when the module is found and successfully loaded or if there are no more directories.

`maxstep` defines the number of instructions to be executed before the program stops with error. This can be useful to limit CGI programs to run too long and consume too much memory and processor. That can happen either by error or because your programmer (customer in case you provide web hosting service) wants to degrade your service.

Note that this limit is compared against the number of steps, which is more than the number of lines. For example the line

```
a = b+3
```

executed by two steps: an addition and an assignment command.

If the value of this parameter is zero or if the parameter is not defined in the configuration file no limit is effective.

`maxlocalstep` defines the number of instructions allowed to be executed inside a function or subroutine before the program stops with error. This is limit is similar to `maxstep` and can serve the same purpose. This may help you fine tune program behavior. Usually setting `maxstep` is enough.

If the value of this parameter is zero or if the parameter is not defined in the configuration file no limit is effective.

`maxlevel` defines the maximal level of recursive call deepness. Programs usually get into infinite loop because of incorrect recursive loop. This value can limit the maximum number of function calls nested. The binary version of ScriptBasic is shipped with a value appropriately set for this value. If you compile the program from source the installation process sets this value.

If the value of this parameter is zero or if the parameter is not defined in the configuration file no limit is effective.

`maxmem` defines the maximum number of bytes that a basic program is allowed to use for its variables. The actual memory allocated by the process or thread running the interpreter may exceed this value because memory for the reader, lexical analyzer, syntax analyzer, code builder and external modules are not limited. This limits only the memory allocated for the ScriptBasic variables.

If this value is zero or is missing then there is no limit except the limits of the operating system and the hardware.

`preload` defines the external modules that are always to be loaded before starting the basic program. There can be more than one modules defined. In this case the modules are loaded in the order they are specified.

The name of the module should be specified the same way as in the ScriptBasic statement `declare sub`: either with full path including file extension or without path and extension. If only the name of the module is specified the module directories are searched by ScriptBasic to load the module defined by the configuration key module.

`cache` defines the directory where the program cache is stored. This should be an existing directory name including the final `\` or `/`. At each run the interpreter will check in this directory if the file was already compiled and a precompiled version is used if that is newer than the file.

`preproc` defines the external and internal preprocessors.

`extensions` Defines the extensions that external preprocessors are assigned to. Each key has to mach an extension and the value string has to be the symbolic name of the external preprocessor assigned to that extension.

`external` Each key in the sub-configuration is the symbolic name of an external preprocessor and the sub-configuration assigned to that key defines the directory where to store the result of the preprocessor and the command to start the preprocessor. Note that a space and the source file name are automatically appended after the command line.

`internal` Defines internal preprocessors. Each key in this list defines the name of the preprocessor and the value defines the DLL or SO file that implements the internal preprocessor. The key name should be used following the option '`-i`' or after the BASIC source program command `USE`.

ScriptBasic ignores all lines containing a key that it does not understand. This is dangerous on one hand because it makes typing errors less recognizable. On the other hand it allows different variations share common configuration file. The external modules implemented as dynamic load library functions can also access any configuration data.

An example configuration file from a Windows NT installation:

```
; scriba.conf
; ScriptBasic sample configuration file
;
; Note that this configuration file format is from v2.0b1 or later and has to be co
; to internal binary format before starting ScriptBasic

; this is the extension of the dynamic load libraries on this system
dll ".dll"

; where the modules are to be loaded from
module "d:\\MyProjects\\sb\\modules\\"
module "d:\\MyProjects\\sb\\sysmodules\\"
module "c:\\ScriptBasic\\modules\\"

; where to search system and module include files
; trailing / or \\ is needed
```

```
include "d:\\MyProjects\\sb\\source\\include\\"
include "c:\\ScriptBasic\\source\\include\\"


;
; define external preprocessors
;
preproc (
; extensions that preprocessors are to be applied on
  extensions (
; here the key is the extension and the value is the symbolic name of the external
; preprocessor
     heb "heb"
     )
; the external preprocessors
  external (
    heb (
      executable "D:\\MyProjects\\sb\\Release\\scriba.exe d:\\MyProjects\\sb\\sourc
      directory "d:\\MyProjects\\sb\\hebtemp\\"

      )
    )
  )

;
; LIMIT VALUES TO STOP INIFINITE LOOP
;

; the maximal number of steps allowed for a program to run
; comment it out or set to zero to have no limit
maxstep 30000

; the maximal number of steps allowed for a program to run
; inside a function.
; comment it out or set to zero to have no limit
maxlocalstep 0

; the maximal number of recursive function call deepness
; essentially this is the "stack" size
maxlevel 300


; the maximal memory in bytes that a basic program is allowed to use
; for its variables
maxmem 1000000

;
```

```
; ScriptBasic loads the modules before starting the code
; in the order they are specified here
;
;preload "ext_trial"

;
; This is the directory where we store the compiled code
; to automatically avoid recompilation
;
cache "d:\\MyProjects\\sb\\cache\\"

cgi (
;

; These are the keys used by the CGI module
;
  debugfile "d:\\MyProjects\\sb\\cgidebug.txt"
  )
```

Although Windows NT programs are usually configured using the registry or active directory services ScriptBasic uses a text file to ease and maintain compatibility with UNIX versions.

To ease the compatibility even more the file names are allowed to use the UNIX style forward slash `/` as directory separator character.

When the program starts one of its first action is to search the configuration file. The search for the configuration file is different on Windows NT and under UNIX operating system.

3.5 Configuration file location under Windows NT

Win32 installation first tries to open the file '`scriba.conf`' if it exists in the same directory as the executable file. This is a convenient place to store the configuration file and does not require registry editing to install ScriptBasic. This is also the ultimate place for CD ROM products utilizing ScriptBasic that need running instantaneously without any installation.

If the file '`scriba.conf`' does not exist in the directory of the executable the configuration manager tries to open the file, which is specified in the string value, named `config` under the registry key `HKEY_LOCAL_MACHINE\Software\ScriptBasic`. If there is no name specified in this registry value ScriptBasic tries to locate the configuration file '`scriba.ini`' in the system directory. The system directory is determined reading the environment variable *windir*. If this environment variable does not exits ScriptBasic tries *systemroot*. It should usually exist on normal Windows installation. If ScriptBasic can not find even this environment variable it tries '`C:\WINDOWS`' as final try. If no configuration file can be found ScriptBasic tries to execute the program without configuration information.

Note that if for example '`C:\WINNT\scriba.ini`' exists and is valid, but the registry defines a different and a non-existent or invalid file ScriptBasic will fail to load the configuration file. It will try to read the file specified in the registry. The other file options are searched when the registry key does not exist or is empty.

If the command line uses the option '`-f`' then the argument of the option is used as configuration file and this overrides all configuration search algorithm. If the file specified in the option '`-f`' does not exists then ScriptBasic runs without configuration.

3.6 Configuration file location under UNIX

UNIX installations try to locate the configuration file from the environment variable *SCRIBACONF*. This environment variable should contain the configuration file name. If this environment variable does not exist ScriptBasic tries to load the configuration data from the file '`/etc/scriba/basic.conf`'.

Note that if the file defined in the environment variable *SCRIBACONF* is non-existent or is invalid ScriptBasic does not try to load the configuration from the default configuration file.

If you can not configure ScriptBasic configuration file name using environment variables or the system registry under Windows NT you can modify the source file '`scriba.c`'. You have to alter the function `scriba_LoadConfiguration`.

If the command line uses the option '`-f`' then the argument of the option is used as configuration file and this overrides all configuration search algorithm. If the file specified in the option '`-f`' does not exists then ScriptBasic runs without configuration.

3.7 Overriding the default configuration file

On both UNIX and Windows NT you can specify a configuration file on the command line. If you specify the file name of the configuration file on the command line following the option '`-f`' then the specified configuration file is going to be used instead of the default.

3.8 Trouble shooting installation

Although installation is quite simple there can be some pitfalls that we and our users have experienced during installing ScriptBasic. To help you we list the possible error.

Symptom Some programs do not run, but claim that the binary version was created for a different build of ScriptBasic.

Cause and solution There is binary cache file for the programs, which is younger than the source of the basic programs but was created by a previous installation. Delete the content of the cache directory.

Symptom Apache (or IIS) claims internal server error when running CGI scripts.

Cause and solution There can be several causes. The most probable ones:

You forgot to start the program using `#! /usr/bin/scriba`

The basic program is not executable by `nobody` (or the web daemon user)

The code of ScriptBasic '`/usr/bin/scriba`' is owned by root and is not executable by the user `nobody`.

The configuration file contains invalid data.

The file names in the configuration miss the trailing `/`

The Windows NT registry string `HKLM\Software\ScriptBasic\config` is missing or points to an invalid configuration file.

4 Code caching

The interpreter has to do several things before actually starting the BASIC code. It has to read the file, split it into tokens, analyze the program and build the internal executable format. This takes a lot of time especially for programs longer than a few lines. This can be skipped if the code built up is saved into a file and the file is loaded and run without the recompilation.

To save the binary format file after the BASIC program was analyzed and build the '`-o`' option can be used. To load and run the code the binary file can be used instead of the text file containing the BASIC code. The interpreter automatically recognizes the binary format and does not try to interpret it as BASIC program source. To help the interpreter you can also supply the option '`-b`' on the command-line that tells the interpreter to believe without checking that the file is binary format of the basic program. This is one solution to speed up execution of the ScriptBasic programs and to avoid recompilation of the code several times. The other method is caching.

The configuration file (see section See ⟨undefined⟩ [Installation Instructions], page ⟨undefined⟩) may contain a value for the key `cache`. The value for this key should specify a readily existing directory (do not forget the trailing `/` or `\`) where the BASIC interpreter can store the binary format of the code. If there is a valid value specified as cache directory the interpreter checks the cache for an already existing binary format file and if that is newer than the BASIC source it neglects the BASIC source and loads the binary format from the cache.

On the other hand whenever a BASIC source code was analyzed and built the resulting binary code is saved into the cache. This method is transparent for the user and significantly speeds up program execution especially for CGI programs that run many times.

To avoid cache using you can use the command-line option `-n`.

Caching has some limitation that can result unwanted behavior. Before switching on program code cache, please read the following sections that describe the shortages and the security issues that should be considered.

4.1 How cache file name is calculated

The cache file name is calculated from the source file name. The applied algorithm calculated the MD5 digest of the source file name and converts it to a 32-character string. The digest itself is 16 bytes long. The conversion to ASCII takes the 32 half-bytes. Each half byte can have a value from zero to 15. Zero is converted to `A`, 1 to `B` and so on. This method was introduced in ScriptBasic v1.0build18 and eliminates some shortages of the algorithm of former versions.

4.2 Cache handling shortages

The cache directory contains the cached binary format program codes for the BASIC programs executed. For each BASIC program a new file is generated when the BASIC source code is analyzed and built.

The file name for the cache file is calculated using the file name. This file name includes the full path to the file under Windows NT but is the bare file name as supplied on the command line under UNIX.

This means that if you have two different programs named '`hello.bas`' in two different directories and you run one after the other from their directory issuing the command line under UNIX

```
scriba hello.bas
```

the second execution will erroneously think that the cache file created when the first one was running belong to this program and will execute the binary code.

On the other hand if you execute the program '`hello.bas`' from two different directories and the supplied paths are different the caching mechanism will not realize that the two execution refer to the same file.

Also note that cache handling does not compare the modification time of the files that basic program includes. If the program itself does not change, but an included file did the caching it will still run the older version.

Caching usually works and in case you experience mysterious problems you can try to avoid caching using the option '`-n`' or commenting out the configuration line `cache`. Web servers usually specify full path to the executed code and in that case cache file name conversion is correct. Especially Apache does start CGI programs passing the full path to ScriptBasic as argument and thus caching works well with this web server.

It is recommended to switch the caching totally off during development and switch on at the start of the test phase. On test environment delete the cache frequently whenever you experience mysterious errors that you think you have already corrected.

4.3 Cache security

Caching is great, but there are some security issues that you have to understand before allowing caching. The ScriptBasic interpreter is usually available to all users in an installation. The cache directory should be created for the interpreter and the security of the directory should be set so that each user can read and write the directory. This may impose security risks in a shared environment.

5 Compiling BASIC programs

BASIC programs can be compiled to a tokenized form, to C code or to an executable file.

The tokenized format is the internal format of ScriptBasic and this is always generated before executing a BASIC program. To speed up execution ScriptBasic is capable saving this code into cache files automatically. However in some situation you may want to save this format into a separate file. To do this you have to issue the command

```
scriba -no myprogram.bbf myprogram.bas
```

This will save the code into the file '`myprogram.bbf`'. After having this file you can execute the BASIC program issuing the command

```
$ scriba -b myprogram.bbf
```

or

```
$ scriba myprogram.bbf
```

On UNIX you can make the compiled file to be executable setting the permissions

```
$ chmod u+x myprogram.bbf
```

and after that you can just write:

```
$ ./myprogram.bbf
```

Please note that binary format files may not be executed on different versions or builds of ScriptBasic and are not movable from one platform to another. You may be lucky to execute Windows NT and Linux version `bbf` files on the other platform, but this is not a guaranteed feature. You surely can not run the binary format generated on a DEC OSF/1 (ooops, sorry Tru64 UNIX) on a 32 bit system.

5.1 Binary format of the BASIC code

In this section we describe some features of the binary format of the BASIC programs saved by ScriptBasic automatically into the cache directory or to a file using the option '`-o`'. There is absolutely no need to know or understand the binary format of ScriptBasic for those who want to program in ScriptBasic. However advanced users may want to perform some modification on binary format BASIC programs, like changing the interpreter location in the start line. For those this section describes the details.

The binary format is binary. It starts with some constant leading code to ensure that no erroneous execution starts and to avoid memory corruption. This leading code helps ScriptBasic to recognize the binary format even if the interpreter was started without specifying the option '`-b`'.

The binary format *may*, however start with a textual line specifying the interpreter. This is needed to make the binary format file executable on UNIX, for example to run a CGI program. This line is interpreted by the UNIX operating system and is ignored by

the interpreter. Because this line has no meaning on Windows NT the line HAS TO be terminated by a line feed and not CR/LF. You need not worry about it, because this is created this way on Windows NT as well as on UNIX. This first line is copied from the source BAS file when the binary format is created.

You may want to change this line when you want to run the code on a machine having the interpreter located at a different location. To do this you can use the following sample ScriptBasic code:

```
cmdlin = command()
split cmdlin by " " to FileName,Interpreter
open FileName for input as 1



binmode 1
File$ = input(lof(1),1)
close 1
if left(File$,1) = "#" then
  i = 1
  while i < len(File$) and mid(File$,i,1) <> "\n"
    i = i+1
  wend
  if mid(File$,i,1) = "\n" then
    File$ = "#!" & Interpreter & mid(File$,i,len(File$))
  end if

  open FileName for output as 1
  binmode 1

  print#1,File$
  close 1
end if
```

This leading line is optional.

The first byte of the binary code (following the optional command line) is the ASCII number of the size of a long on the actual platform. This is currently 4 on Linux and Windows NT, Windows 95/8 and it is 8 for Tru64 UNIX.

This is followed a magic code. This is `0x1A534142`. On DOS platforms this is printed as `BAS^Z` and prevents dumping the code to the screen if one attempts. This magic code is saved as a `long`. This means that the order of the bytes follows the order of the bytes in a long in memory on the machine the code was saved. This is followed by six long values. These are

`VersionHigh` the high number of the ScriptBasic version

`VersionLow` the low number of the ScriptBasic version

`MyVersionHigh` the high number of the variation version

`MyVersionLow` the low number of the variation version

`Build` the build number of the version

`Date` coded date

The following eight bytes contain the name of the variation that created the binary code. Because some variations do not differ in binary format from the STANDARD variation they create binary file saying it being STANDARD.

To successfully load a binary format file to run in ScriptBasic the long size, the magic code, version information including the build and the variation string should match. Date may be different.

Following this header the real information of the executable code is placed. The following four long numbers present in order the

Number of global variables

The number of nodes in the code

The start node serial number

The size of the table containing the constant strings

Following this the nodes come one after the other. ScriptBasic prints the size of a single node in bytes when the option '`-v`' is used. This is usually 16bytes on 32bit systems and 24bytes on 64bit systems. If you get smaller node size printed then you will face alignment problems. If you get bigger node check your compiler options. ScriptBasic was not tested on 128bit systems up to now.

The string table follows the nodes. This contains all the strings zero character terminated that present in the code. There is at least a zero valued byte in the string table even if the program does not contain any string constant.

There can be arbitrary data following the string table, they are ignored.

6 Compiling BASIC program to EXE

ScriptBasic program can be "compiled" to executable directly under Windows and Linux. To do this you should use the option '`-E`' and you should also specify an output file using the option '`-o`'. For example:

```
scriba -Eo queens.exe queens.bas
```

will generate an executable file '`queens.exe`' that will run the BASIC program '`queens.bas`'. When using this option the ScriptBasic interpreter will ignore any cached precompiled code and will compile the BASIC program from source code fresh.

Note that this kind of compilation works only on Linux and Windows and the generated file is operating system specific, and can not run on any other operating system. Beta test users reported that the feature works on FreeBSD and on Solaris, but the developers lacking test resources can not guarantee this.

Although the ScriptBasic interpreter compiled and installed on other operating system (above Windows and Linux) will accept these options and will generate a file it may not be usable. This feature uses the specialties of the executable format of Windows and Linux.

The compilation to executable is not a real compilation, in the sense that the program is not compiled into real machine code, and thus do not expect it running faster or using less memory. This methodology only creates the intermediary code that the ScriptBasic execution module uses and puts it into an executable file that also contains the interpreter. The final executable will contain the binary code of the BASIC program and will execute the same way as it would execute running the original file.

In the rest of this chapter we detail how the BASIC to EXE direct compilation works.

6.1 BASIC to EXE compilation technical details

Under Windows and Linux an executable file is loaded into the system without caring too much the length of the file. This is the feature that let viruses attach themselves to executable file under Windows. When ScriptBasic is started with the option '`-E`' the interpreter makes a copy of the executable of itself and appends the compiled intermediate code after the executable code. Finally it prints the eleven characters string `SCRIPTBASIC` and a four byte file byte offset pointing after the end of the executable part of the file.

When the interpreter starts first it check if the last 15 bytes start with the string `SCRIPTBASIC`. If it does not then the interpreter continues in normal operation processing the command line parameters. However if the last 15 bytes start with the string `SCRIPTBASIC` then the executable is a ScriptBasic generated executable and it loads the code from the executable starting at the byte offset specified by the last four bytes of the file.

7 Compiling BASIC programs to C

Although ScriptBasic is an interpreted language it is possible to create stand-alone executables from BASIC programs on any platform where ScriptBasic can run. To get an executable standalone program first you have to compile your BASIC program to C and the C code has to be compiled to get the executable. (Unless you use the option '`-E`' that can generate executable directly.)

To get the C version of the BASIC program you should use the command line option C, for example:

```
scriba -nCo myprogram.c myprogram.bas
```

This command will compile the BASIC program '`myprogram.bas`' and save the code in C language format into the text file '`myprogram.c`'. You have to compile this file using a C compiler and link it with the library files.

When using this option the ScriptBasic interpreter will ignore any cached precompiled code and will compile the BASIC program from source code fresh. When compiling BASIC to C do not forget to specify the output file using the option '`-o`'. Without this option ScriptBasic does not perform the compilation, there is no default file name for the output.

Note that this is not a real compilation. This methodology only creates the intermediary code that the ScriptBasic execution module uses and puts it into a C program. The final executable will contain the binary code of the BASIC program and will execute the same way as it would execute running the original file.

The achievements you can get compiling the code and generating standalone executable file are:

Faster startup, because the executable does not need to compile or even load the BASIC program, but can execute it immediately. This is a bit even faster than the EXE file generated by directly compiled from BASIC.

Simplified setup of your program, because you only have to deliver a standalone exe to your users.

You can hide that your program was written in BASIC and can a bit protect your intellectual property.

Also note that you have to have a properly created configuration file and the `dll` or `so` modules on the system your executable is going to run in case you need some features that rely on configuration settings.

You may also consider recompiling the ScriptBasic interpreter to statically link some of the external modules that your BASIC program uses.

7.1 Compilation under Windows NT

Currently Visual C command line `cl` and Borland command line `bcc32` compilers are supported.

There are two command files that you can find in the `bin` directory under the installation directory. These are '`bascl.cmd`' to automate the compilation using the Microsoft compiler and '`basbcc32`' to automate the compilation using the Borland compiler.

The '`bin`' directory also contains the files '`libscriba.lib`' and '`libscriba.obj`' to be used by the Microsoft compiler and linker, as well as '`libscriba_omg.lib`' and '`libscriba_omg.obj`' to be used by the Borland compiler and linker.

The content of the file '`bascl.cmd`':

```
scriba -n -Co %1.c %1.bas
cl /Ox /GA6s /DWIN32 /MT /nologo /W0 /c /Fo%1.obj %1.c
cl /Ox /GA6s /DWIN32 /MT /nologo /W0 /Fe%1.exe %1.obj \ScriptBasic\bin\libscriba.ob
```

The content of the file '`bascbcc32.cmd`':

```
scriba -n -Co %1.c %1.bas
bcc32 -5 -c -o%1.obj %1.c
bcc32 %1.obj \ScriptBasic\bin\libscriba_omg.obj \ScriptBasic\bin\libscriba_omg.lib
```

This is the content as the files are installed. In case you have more than one disk, or the installation directory is not '`\ScriptBasic`' you may need to edit the file you intend to use replacing the `\ScriptBasic\bin\` with the actual directory where the object and library files are. It is recommended that you also edit the disk drive letter into the full path so that the command script works in case the actual working directory is on a different disk than the one where ScriptBasic is installed.

The reason that there are two different object and library files for the two different compilers is that these compilers support different object and library format. Microsoft supports the COFF format, while the free compiler available from Borland supports only OMG format. Although there is a `coff2omg` converter available from Borland that works only for library files that are DLL wrapper libraries and not for object files or for library files containing actual code.

These object and the library files contain the same code with the exception that they were created using different compilers.

7.2 Compilation under UNIX

The UNIX installation puts '`libscriba.a`' in the directory '`/usr/local/lib`'. This means that you can use the `gcc` option '`-l`' to specify this library. Therefore the command line to compile the generated C file is:

```
$ gcc -o myprogram myprogram.c -lm -ldl -lpthread -lscriba
```

This will compile the ScriptBasic generated '`myprogram.c`' to the executable '`myprogram`'.

8 Compiling ScriptBasic with modules

This section is rather technical and not for the BASIC programmer. This section details how to compile the ScriptBasic interpreter so that it already includes some of the external modules. This requires knowledge how to compile C programs and some programming practice of C is also helpful. If you just want to program in BASIC do not read this section and get distracted by this, skip to the next section.

ScriptBasic supports external modules that are usually delivered as DLL or SO files and using them do not require the recompilation of the interpreter. However in some cases there is a need to compile the ScriptBasic interpreter and some of the external modules into a single executable image. This may be the case when you deliver a solution to a customer and want to build a special version of the interpreter lessening the possibility of misconfiguration or you just simply want to protect your intellectual property implemented in the external module and do not want to deliver it as DLL or SO file. Linking some modules static into the interpreter also provides some speed improvement.

To link some modules into the interpreter you have to create a file named '`lmt_XXX.def`'. This file should list the names of the modules that you want to statically link. There can be empty lines and lines starting with the character `#` as comment. You can use the characters `XXX` (not necessarily three character, but try to avoid the word `httpd` and `none` as we have already have those) to distinguish your compilation.

After the file is created issue the command:

```
perl lmt_make.pl lmt_XXX.def
```

This will create the file `lmt_XXX.c`.

THIS IS CHANGED TOTALLY FOR v1.0BUILD30 and ABOVE. TO BE REWRITTEN ALTER. ACTUALLY ALL THESE THINGS SHOULD GO INTO THE DEVELOPERS GUIDE!!

If you look into the file '`lmt_XXX.c`' you will see that the module tables are referenced by the global table *StaticallyLinkedModules*. You have to give the name of the function as a string as well as the function as well. This table will be searched by ScriptBasic when the module is "loaded". When a module is loaded from a DLL or SO file such a table is provided by the operating system and is searched by operating system functions.

The next step is to compile the modules for the static linking. This is a bit different from compiling for DLL or SO file as result. When the module is compiled to be linked into a DLL or SO file all interface functions should be exported so that the operating system can access their name and entry point. When the module is compiled to be linked together with the interpreter all functions should be static. If we compiled the module the same way as for targeting DLL or SO file we were not able to link more than one module into the interpreter because the linker would complain about multiple defined `versmodu` and other functions that are defined in each module.

To compile a module for static linking you should use the compilation option '`-DSTATIC_LINK=1`' that tells the C preprocessor to define some macros differently than normal. If you look into the '`Makefile`' or '`Makefile.nt`' you will see that these compilation options are already there and all module interface files can be compiled using the tool make to its static version which is named '`s_module.o[bj]`'. The prefix `s_` is

used to denote that this version of the object file was compiled from the same C source using the options that are required for static linking of the interpreter.

Choose an appropriate executable name `XXX` for your version and edit the `Makefile`. Create the rule that compiles your version. Your version will eventually use '`lmt_XXX.c`' and '`lmt_XXX.o[bj]`' instead of '`lmt_none.*`' and the executable will depend on the source and should be linked with the compiled object files and libraries of the modules. As an example you can look at the variation `sbhttpd` that includes the CGI and the MT module linked into it.

Type

```
make XXX
```

And you should get your executable.

9 General Language Format

This chapter defines the language format, how to program in BASIC. The later chapters define in detail the functions available in the interpreter.

9.1 Hello World

ScriptBasic is a BASIC language. As such it is line oriented. The commands follow each other in the source file lines. The simplest example, the usual "hello word" example is a one-liner in basic:

```
print "HELLO WORLD"
```

You can write the commands upper or lower case or even mixed case. Therefore the following lines are equivalent:

```
Print "HELLO WORLD"
print "HELLO WORLD"
PRINT "HELLO WORLD"
PrInt "HELLO WORLD"
```

9.2 Strings

Strings are most frequently used entities in a BASIC program. As you could already see in the very first example strings are quoted with double quote. But they can be more than just characters between double quotes. String as a type is a basic data type of ScriptBasic. A variable having string value contains several bytes. There is no length limit for a string other than the virtual memory of the machine. A string may contain character of any code even characters that have the ASCII code zero. In other words a string is a collection of bytes of certain length. You can treat strings as arbitrary length of binary data if you need.

Strings can be concatenated, you can cut off a part of a string and several other functions and operators can handle strings. ScriptBasic automatically allocates the space required to store the string and releases the unused space when the string is not needed anymore. For more information on string handling operators and functions see the chapter See ⟨undefined⟩ [Command reference], page ⟨undefined⟩.

The simplest form of a string is a strings constant appearing in the source file. This is the form like in the example

```
print "Hello Word\n"
```

Here you can note that there is a special character at the end of the string denoted by two characters. This may be familiar from other languages. The characters `\n` mean a new line character in a readable form. The `\` (backslash) character is the escape character in a string altering the meaning of the character that follows it. The special characters that ScriptBasic handles are:

`\t` is converted to a tabulator character.

`\n` is converted to a new line character.

`\r` is converted to a carriage return character.

`\"` is converted to a double quote character. This is the way to include a double quote character into the string.

`\0-9` is converted to ASCII code, see details later.

All other characters remain the same after a backslash. This means that you can write `\\` to have a string containing a backslash character, or `\"` to have a string containing a double quote character, but `\a` means nothing else than the letter `a`.

There is another way to include special characters into string constants. The usual way in BASIC is to split the string into sub strings and concatenate the parts during run time, like in the code fragment:

```
St = "This is a special string, containing a bell character at the end" & chr(7)
```

You can do this in ScriptBasic, but you can do it easier and more effective:

```
St = "This is a special string, containing a bell character at the end\7"
```

The last character is a number preceded by a backslash. Whenever numbers follow a backslash character in a string ScriptBasic calculates the value of the numbers and uses the character of the code. If the first character after the backslash is zero then the number is treated as octal number, otherwise it is treated as a decimal number.

Strings should not contain the new line character. In other words

```
St = "this is
a multi line
string of three lines."
```

is not legal in ScriptBasic. It was legal in former versions before v1.0build15 to write multi-line strings, but it caused problem to programmers forgetting the closing double quote character on a line. Instead a new string constant format was introduced that starts and ends with three double quote characters. For example:

```
St = """this is
a multi line
string of three lines."""
```

is perfectly correct. Strings starting and ending with three " characters can, but need not span multiple lines. There is another difference between single-line and multi-line strings. A multi-line string may contain a double quote character without escaping with backslash. You can write:

```
St = """this is " a double quote character """
```

or you can write

```
St = """this is \" a double quote character """
```

which is also correct. The only situation where you should escape a double quote character in a multi-line string is when you want to have three or more " characters following each other in a string. For example

```
St = """this is """ three double quote characters """ THIS IS WRONG
```

is wrong. You have to type instead:

```
St = """this is \""" three double quote characters """
```

or

```
St = """this is "\"" three double quote characters """
```

or

```
St = """this is ""\" three double quote characters """
```

or

```
St = """this is \""\" three double quote characters """
```

or some other variation of escaping one or more of the consecutive " characters. The rule is that there can not be three consecutive un-escaped " characters inside a multi-line string.

There is another constrain regarding multi-line strings. The very first character of a multi-line string should not be the character & or it has to be escaped with the backslash character. Thus:

```
St = """\& is a
multi-line string starting with an & character """
```

is the correct format. If you use the & character without escaping it as the very first character of a multi-line string it will be treated as a binary multi-line string.

A binary multi-line string is a very special beast that only hard code users need to specify binary data inside a BASIC program. In such strings all new-line characters are ignored unless escaped with the back-slash character. Thus the following two strings

```
"""&is a
binary multi-line\n string starting with an & character """

"is a binary multi-line\n string starting with an & character "
```

are equivalent with the exception that the second string is a liar, because that is not a binary multi-line string. Single-line strings can not be binary, and in case the first character of a single-line string is the & character that is just treated as a normal character.

9.3 Numbers

There are two types of numbers in ScriptBasic:

integer numbers and

real numbers.

The integer numbers can be used to represent integral values, while real numbers can be used to represent number that have fractional part or are too large to store as integer. The integer numbers are stored in a memory location of size equivalent to a `long` of the programming language C. The real numbers are stored internally as C `double`.

Number constants can be used in the basic program in the usual format. Integer numbers are represented in either decimal or hexadecimal format. Decimal numbers contain only digits. Hexadecimal numbers start with the characters `0x` or `0X` and are followed by hexadecimal digits. The format that many basic implementation follows using the `&H` characters to start a hexadecimal number is also allowed. When a number contains a `#` character inside, like `2#110111` then the number preceding the `#` is the RADIX of the number and the characters following the `#` is the number in the given radix. Because the limited number of characters in the ABC the RADIX can go from 2 to 36 only. The following numbers are valid integer constants in ScriptBasic:

`123`

`0xFF` equals to decimal `255`

`0x255` equals to decimal `597`

`0X0` is a hexadecimal zero

`&H52` equals to decimal `82`

`17#GG` is 288

There is no internal difference between decimal and hexadecimal numbers for ScriptBasic. The lexical analyzer converts both format to internal representation and stores the value of the number. In other words wherever you are allowed to use a decimal number you are allowed to use a hexadecimal number or any radix number as well. You should decide whether to use decimal or hexadecimal or any other radix number for your convenience taking care of BASIC source code readability.

Real number constants can only be decimal and may contain fractional and exponential part. The followings are valid real number constants:

```
3.14
1.0
1.
2.3E-7

2.3e7
2.3E+7
```

Note that the exponent part is preceded by the character `e` or `E` and is followed optionally be a sign. The number `1.` is a real number and is stored in a C `double` internally although it could be integer. ScriptBasic will store this number as a real number. However real numbers are automatically converted to integer values whenever ScriptBasic needs an integer value.

9.4 Variables

Variables are core entities of ScriptBasic. Variables are used to store string, real or integer values. Variable names start with alpha characters, underscore, dollar sign or colon, and from the second character they may contain digit characters in addition to all these characters. The last character of a variable name should not be colon.

Whenever you need a variable choose a meaningful name. If you are a real old BASIC programmer use trailing `$` to denote string variables. However ScriptBasic can store any value in any variable, one at a time of course.

The colon as a name character is allowed to help name space management, and you should use it only for the purpose.

The following are valid and invalid variable name examples:

```
myvariable   This is a perfectly legal variable name.
main::var    This is OK. This variable is in the name space main.
chr$         This is invalid. Chr$ is a reserved word,
             this is a built-in function
apple$       This is OK. You will use it probably to store strings.
```

```
b:2          This is valid, but it is recommended not to use : inside
             variable names, because it is unreadable.
             Use colon only to separate hierarchical name space and
             variable name.

_mother      If it is your taste to start a variable with underscore you can.
Beee$bop     Valid, but it is not recommended. The dollar sign is allowed in
             variable names to allow the usual BASIC string variable notation.
             System or application specific extensions may use predefined global
             variable names that contain a $ sign inside. Using such variable
             names you may get into conflict.
::boo        This is valid. This variable is explicitly noted to be in the
             current name space.

_::baa       This is valid. The variable is in the parent name space.
::chr$       This is valid, even though chr$ is a predefined function.
```

For more information on name spaces read the chapter See ⟨undefined⟩ [Name spaces], page ⟨undefined⟩.

Variables can contain any data in ScriptBasic. There is nothing like integer or string variable. A variable may contain integer value at a time a real value another time and string value later. You can use a variable name to use real, integer or string value at a time; later you may use the variable as an array; later as a real again. You can change it any time.

Variables can be local or global. Global variables are those that are not declared but used (unless `declare option DeclareVars` is specified in the code, or `declare option DefaultLocal` declaration is in effect). Any variable by default is global unless it is declared to be local. Local variables are local to the function or subroutine in which they are declared to be local. To declare local variables you should use the command `local`. (Note that it is possible to use the directive `declare option DeclareVars` to require explicit variable declarations. This directive is detailed later.)

```
REM This is a sample program to demonstrate local variables


'A is a global variable
A=13
Call MySUB
Print A,"\n",B

Sub MySUB
Local A
A=9
B=55
End Sub
```

The output of the program is

```
13
```

```
55
```

This is because the variable *B* inside the sub is global, but the variable *A* is local and as such it does not alter the value of the global variable *A*. You can define one or more variables to be local in a local statement. If you declare more than one variable to be local then you have to separate the variable names with commas.

9.5 Constants

Constants are values that the program can not alter. Unlike variables constants have a single value and this value does not change during program execution. The strings and numbers that are used in a program are naturally constants. They are the simplest forms of constants without a name. However you may want to name constant values. You could store the value in a named location, in a variable but that consumes an extra variable and slows down execution.

ScriptBasic allows you to define constants using the format:

```
const name=value
```

or

```
global const name=value
```

where *name* is an identifier and value is a string or a number. You can not specify an expression as a value, only constant string or number. Later you can use the name of the constant in any expression where you could use the value of the constant. If you have the constant declaration:

```
Const nl = "\n"
```

then the following two lines are identical, they create the same executed code and therefore execute the same speed and need the same size of memory:

```
print 6+6,nl
print 6+6,"\n"
```

ScriptBasic evaluates named constants during syntax analysis and the name of the constant is replaced by their defined value in the expression where the constant is used.

Named constants are local to the subroutine or function they are defined in unless you use the keyword `global`. If you declare a constant in a module (see chapter See ⟨undefined⟩ [Name spaces], page ⟨undefined⟩ for more on modules and name spaces) the named constant becomes part of the name space and you can refer to the constant only with full name space specification from outside of the module. If you declare a named constant inside a function or a subroutine without the keyword `global` the constant becomes local, and can not be used outside of the subroutine or function.

If you use the keyword `global` the constant is declared global. This means that the constant is not part of any name space and can be used anywhere in the program after the line it was declared.

When ScriptBasic finds an identifier in an expression, which is not a built-in function or reserved word it checks the followings until a check succeeds:

Checks if the identifier is a local constant of the current function or subroutine. This check is skipped if the expression is inside of any function or subroutine.

Checks if the identifier is a constant of the current name space.

Checks if the identifier is a global constant.

Checks if the identifier is a local variable of the current function or subroutine. This check is skipped if the expression is not inside of any function or subroutine.

If all above checks fail the identifier is treated as global, module specific variable.

Constants can be redefined. When a named constant is defined in a const statement ScriptBasic does not check if the constant has already be defined. It is legal to change the actual value of a constant during compile time. Note however, that this does not make a constant to be a variable. The change of the value is performed during syntax analysis and not during execution. `Const` statements do not generate any executable code and therefore are never executed.

When an identifier is declared as a constant the identifier can not be used as variable in an expressions following the program line that defines the constant. Sometimes you want to delete a defined constant and use the identifier as variable again. To use an identifier as a variable that was already defined as constant you can use the command

```
var name
```

where the `name` is the identifier. This declaration tells ScriptBasic that the identifier is not a constant anymore. This declaration is similar to the normal, non-global const declaration. If you write a `var` statement inside a function or subroutine the identifier becomes a local or module specific global variable in the function or subroutine only. If you write a `var` statement outside any function or subroutine the identifier becomes a variable in the module only.

To explain the behavior let us see an example:

```
module TEST
a = "GLOBAL VARIABLE"
Global Const a = "a "

Const b = "b "

sub TestSub
  const c = "c "
  ' all three identifiers are constants here
  print "  values in TestSub=",a,b,c,"\n"
  ' from now on until end of the sub
  ' 'a' is a variable (global or local)
  var a

  ' here 'a' is a global variable
  print "a in sub as a global variable holds the value=",a,"\n"


  ' now a is defined to be a local variable from now on
  ' until the end of the subroutine
  local a
  a = "LOCAL VARIABLE"
```

```
  print "a in sub=",a,"\n"
end sub

' 'a' and 'b' are constants here again as
' 'var' declaration was inside the sub
print "values in the module=",a,b,c,"\n"

print "values called from within the module:\n"
TestSub

' 'a' is a variable from now on (global only
'                                as we are global here)
var a

' here a is a global variable again
print "a in module=",a,"\n"

end module

' here 'a' is a global const (var a was inside the module)
' --
' 'b' was const but not 'global const', so here it is
' a global variable again
' --
' 'c' was inside the module, inside the sub. Here it is
' totally undef
print "values outside the module=",a,b,c,"\n"

print "values called from outside the module:\n"
TEST::TestSub
var a
print "a in global=",a,"\n"
```

The output is:

```
values in the module=a b undef
values called from within the module:
  values in TestSub=a b c
a in sub as a global variable holds the value=GLOBAL VARIABLE
a in sub=LOCAL VARIABLE
a in module=GLOBAL VARIABLE
values outside the module=a undefundef
values called from outside the module:
  values in TestSub=a b c
a in sub as a global variable holds the value=GLOBAL VARIABLE
a in sub=LOCAL VARIABLE
a in global=undef
```

To help understand even deeper the behavior of local, global, module specific constants and variables here we present a short description how the syntax analyzer handles the constants. You need read this only if you are curious.

There are several symbol tables during syntax analysis. ScriptBasic maintains a symbol table

for the labels,

one for global variables,

one for local variables and

a separate one for constants.

When a constant is defined the name and the corresponding value gets into this symbol table. If the const statement is global the name is not altered. If the const statement is not global the name is modified to include the name space. This modification is the same as the modification for variables. After this modification an apostrophe character is appended to the constant name and the name of the actual function or subroutine if the constant is defined inside one. This is the same name decoration mechanism, which is performed for the labels.

When an identifier is found in an expression the syntax analyzer searches for the name *module::constname'function* in the symbol table if the expression is inside a function or subroutine. If the expression is in a global area - out of any subroutine or function - this search is not performed. If there is no such entry in the symbol table the analyzer searches for the symbol *module::constname'*. If this symbol is still not defined in the symbol table the analyzer searches *constname*.

If any of the names can be found in the symbol table the identifier *constaname* is replaced in the token list with the value of the constant.

When a `var` statement declares an identifier to be constant it actually does a weird action inside the interpreter. It redefines the constant name `module::constname'` or `module::constname'function` to be associated with the value `NULL`. In other words, when the syntax analyzer finds the constant name in the symbol table and retrieves the pointer that is supposed to point to the constant replacement lexical element it gets a pointer with `NULL` value. But it does find the constant name in the symbol table and the search finishes successfully. On the other hand the next step in the syntax analysis sees only the `NULL` value in the constant replacement pointer and thinks that no constant was found.

This means that if the basic program has a global constant and we declare the identifier to be a variable using a `var` statement the constant search stops when it finds it to be a local (either function or subroutine local or module local) constant. A local constant that happens to have a replacement pointer pointing to `NULL`. And it does not search for the global constant, because it did find a symbol table entry.

The same situation happens if a module constant is redefined using the `var` statement inside a subroutine or function.

ScriptBasic defines some named constants before it starts analyzing the program. These named constants always start with the letters `sb`, and their purpose is to help the programmer to set various options in option statements.

The global constants defined by the ScriptBasic interpreter are:

`sbCaseSensitive`

```
sbCaseInsensitive
sbMathErrDiv
sbMathErrUndef
sbMathErrUndefCompare
sbCollectDirectories
sbCollectDots
sbCollectRecursively
sbCollectFullPath
sbCollectFiles
sbSortBySize
sbSortByCreateTime
sbSortByAccessTime
sbSortByModifyTime
sbSortByName
sbSortByPath
sbSortAscending
sbSortDescending
sbSortByNone
SbTypeUndef
SbTypeString
SbTypeReal
SbTypeInteger
SbTypeArray
```

These constants are defined by the interpreter and there is no need to include any file to use these constants. Other constants are defined in various header files.

9.6 Forcing Variable Declaration

BASIC languages generally do not require variable declaration to be programmer friendly. ScriptBasic is no exception for compatibility reasons: you can write huge programs without declaring any variable. Although this is possible this is not a good practice. Programmers can easily mistype a variable name and end up getting a new variable with `undef` value instead of getting compilation error. This may cost a lot of debugging time.

To help programmers avoid such a situation ScriptBasic can be asked to report any such variable as compilation error. However this also means that the programmer has to declare the global variables. To tell ScriptBasic that a certain part of the program or the whole program requires variable declaration the programmer has to issue the declaration

```
declare option DeclareVars
```

This is a directive that does not generate any code, but tells the compiler to require variable declaration. To declare global variables the program should use the command `global`. For example

```
declare option DeclareVars
global a,b,c
```

declares three global variables *a*, *b* and *c*. The directive `declare option DeclareVars` is effective from the line where the directive is. Any undeclared variable used before the directive can also be used without declaration after the directive as well. For example:

```
a = 2
declare option DeclareVars
global b,c
b = 3
c = 4
print a,b,c
print
```

The variable *a* is implicitly declared before the directive and thus it can be used even after the directive. Before the directive the global variables are implicitly declared. As no global variable is allowed to be declared more than once such an implicitly declared variable should NOT be declared in a `global` declaration. For example

```
a = 2
declare option DeclareVars
global a
```

generates compilation time error. Once you switched on the variable declaration requirement, you should declare each new global variable until the end of the program. This means that all included or imported files should declare all global variables if you use the directive before including or importing a file. Because this may pose some incompatibility with older code you can use the directive

```
declare option AutoVars
```

Following this line the declaration is implicit again until the end of the program or until the next `declare option DeclareVars`. You can switch on and off global variable declaration forcing as many times as you like. It is also not an error to switch it on if this is already on, or off it is already off. The compiler does not count however the number of on and off directives. Thus

```
declare option DeclareVars
declare option DeclareVars
declare option AutoVars
```

is just the same as

```
declare option DeclareVars
declare option AutoVars
```

or just the same as

```
declare option AutoVars
```

It is always the last `declare option` directive, which is in charge.

Although you can mix program segments that require and do not require global variable declaration it is recommended to issue `declare option DeclareVars` at the start of the program and declare all global variables.

9.7 Arrays

Arrays are memory locations that store many values at the same time. While normal variables store a single value at a time, an array variable can store many values. The values are accessed via the name of the variable and the appropriate indices. The index or indices follow the name of the variable between `[` and `]`. This is the usual notation for most programming languages, like C, Perl, PASCAL, Python. Some BASIC implementations use `(` and `)` instead, but that confuses array access and function call.

In the following subsections we describe how to use arrays.

9.7.1 Creating arrays

Any variable can become an array. In ScriptBasic arrays are automatically created. There is no statement like `DIM`. All you have to do is to use the variable like an array. Array subscripts are written using square brackets. This is usually the convention in PASCAL and in C. Other BASIC languages use normal parentheses to index arrays. That confuses the reader as well as the parser, because function call looks the same as array access. Therefore ScriptBasic uses the square brackets.

There is no limit on the number of indices. You can use as many as you like. Also there is no limit on the index values other than the index values have to be integer and that memory may limit the array sizes. Positive, negative or zero integers can play the role of an index. Whenever you access an array element or assign a value to that ScriptBasic automatically checks if the referenced array element exists or not and adjusts the array if necessary. For example:

```
a[1] = 3
a[5]=4
for i=1 to 5
print a[i]
print
next
```

is perfectly legal and prints:

```
3
undef
undef
undef
4
```

Arrays are handled quite liberal. You are not required to declare the index bounds, you need not declare the number of indices. As a matter of fact you can have different number of indices at different points in the array. For example the following code is also legal:

```
a[1] = 3
a[5,3]=4
print a[1],"\n",a[5,3]
```

You can even write:

```
a[1] = 3
```

```
a[5,3,1,6,5,4,3,6,4,3,2222]=4
print a[1],"\n",a[5,3,1,6,5,4,3,6,4,3,2222]
```

if you wish.

What happens if you write:

```
a[1] = 3
a[5,3]=4
print a[1],"\n",a[5]
```

ScriptBasic will print

```
3
ARRAY#008C0BC8
```

or some similar message. What has happened? To understand we have to explain how ScriptBasic stores the arrays.

An array in ScriptBasic is stored as a list of C pointers. When a ScriptBasic variable first time used as an array a new array of a single element is created. It has one element assigned to the index that was referenced.

```
VARIABLE[6] = 555
```

Later, when other elements are referenced the array is automatically extended. For example if the array was first time referenced using the index 6 and it was accessed second time using the index 11 ScriptBasic automatically extends the array to contain six elements.

```
VARIABLE[11] = "a string"
```

This means that an array can consume significant memory even if only a few indices are used.

```
VARIABLE[10,3] =6.24E3
```

When an array element, let's say index 10 is used with a second index, let's say with 3 a new one-element array is created. Later this new array is treated the same as the one assigned to the variable itself, and when the element

```
VARIABLE[10,6] = 55
```

is set it is enlarged to be able to hold all values between indices 3 and 6.

When the variable in our example named VARIABLE gets a new value, like

```
VARIABLE = "NEW VALUE"
```

the arrays are released automatically.

When accessing an array element, which is an array itself ScriptBasic tries to do its best. However the result may not be what you expect.

9.7.2 Array index limits

Because array indices are automatically adjusted the lower and upper limit of indices of an array are not constant. They may change. To get the lowest and the highest index of an array that has assigned value (possibly `undef`) you can use the function `lbound` and `ubound`. The names stand for lower bound index and upper bound index.

For example the simple program

```
a[1] = undef
print lbound(a)," ",ubound(a)
print

a[2,-3] = "whoops"
print lbound(a)," ",ubound(a)
print
print lbound(a[2])," ",ubound(a[2])
print
```

will print

```
1 1
1 2
-3 -3
```

As you can see the argument to these functions can be a variable that has array value or an expression that has an array value. (Such an expression is likely to be an array element, which is an array by itself.)

When the argument is not an array the functions returns `undef`.

9.7.3 Deleting an array

Usually there is no need to delete an array. Arrays are allocated when they are needed and are released when they are not needed anymore. However there may be some circumstances, when there is a need to explicitly delete an array and release the memory assigned to the array. To do this you have to use command `undef`. The name is the same as the function `undef`, but instead of returning the undefined value this command sets the variables listed after the command name to hold the undefined value. The format is

```
undef variable_list
```

The variable list may contain variables that are arrays, variables that hold string, integer or real values and may also contain array elements, like *a[13]*, which is a variable itself. When you apply the command `undef` to a variable, which is not an array or reference the result is the same as assigning the undefined value to the variable. For example the following lines

```
undef A
A = undef
```

have the same effect unless A is an array or reference. However when the variable A is an array assigning to it the undefined value will make the first element of the array undefined and not the whole array. To explore the effects try to run the following program:

```
sub fun(a)
print a,"\n"
print lbound(a),"\n"
print a[1]," ",a[2]," ",a[3]
print
print
end sub

k[1] = 1
```

```
k[3] = 2
fun k
k = undef
fun k
undef k
fun k
```

After running it comment out the line `undef k` and compare the result running the program again.

9.8 Associative Arrays

Associative arrays were introduced in ScriptBasic v1.0build19. This section does not apply to any previous version.

Languages, like Perl or AWK have associative arrays and so does ScriptBasic. Associative arrays differ from arbitrary arrays in two points:

The index is written using curly brackets, like

```
a11
```

instead of normal brackets.

The value of the index can be anything not only integer numbers.

When you use a variable as associative array you can use strings, integers, real numbers and even the value `undef` as index value. Whenever you access certain element of an associative array the interpreter searches for the index value and uses the value associated with it. This leads to compact and easy to read programs.

For example:

```
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
a"dummy" = "dummy value"

print a"foo",nl
```

will print

```
foo value
```

If you use an index that is not present in the associative array the result is `undef`.

```
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
a"dummy" = "dummy value"


print a"FOO",nl
```

will print `undef`.

Note that in this situation a new element with the new key is created automatically containing the value `undef`.

Index values that are `undef`, integers or real numbers can be referenced using the exactly same value. When using strings as keys the matching is performed according to the option '`compare`'. The previous example altered a bit:

```
option compare sbCaseInSensitive
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
a"dummy" = "dummy value"

print a"FOO",nl
```

will find the key and print `foo value`, although the key used for definition and the key used for reference differ is case.

Associative arrays inside ScriptBasic are stored as normal arrays. The difference between normal array and associative arrays are the way the programmer handles them. Advanced programmers may and most probably should use associative arrays as normal arrays as well.

An associative array is nothing else than a normal array storing the keys on the even indices and the values on the odd indices. When you reference

```
a"foo"
```

the interpreter starts to search the string `"foo"` on the even indices of the array and when it finds it returns the next element. To explain it see the following sample:

```
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
a"dummy" = "dummy value"

for i=lbound(a) to ubound(a)
print i," ",a[i],nl
next i
```

will print

```
0 foo
1 foo value
2 bar

3 bar value
4 dummy
5 dummy value
```

You can see that the keys and values are paired up in the array. Knowing this you can iterate over the keys of an associative array:

```
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
```

```
a"dummy" = "dummy value"

for i=lbound(a) to ubound(a) step 2
print "a",a[i],"=",aa[i],nl
next i
```

to get

```
afoo =foo value
abar =bar value
adummy =dummy value
```

Note, however that in case the same key present many times in the array using the `a` notation finds always the first value. For example:

```
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
a"dummy" = "dummy value"
a[6] = "foo"
a[7] = "never found"


for i=lbound(a) to ubound(a) step 2
print "a",a[i],"=",aa[i],nl
next i
```

will print

```
afoo=foo value
abar=bar value
adummy=dummy value
afoo=foo value
```

and it never print `never found`.

Knowing that the associative arrays are just normal arrays inside one can even make dirty tricks:

```
const nl="\n"
a"foo" ="foo value"
a"bar" = "bar value"
a"dummy" = "dummy value"
a[-1] ="hoho"

for i=lbound(a) to ubound(a)
print i," ",a[i],nl
next i

print "--------------------------------\n"

top = ubound(a)
for i=lbound(a) to top step 2
print i," a",a[i],"=",aa[i],nl
```

```
next i
```

will result

```
-1 hoho
0 foo
1 foo value
2 bar
3 bar value
4 dummy

5 dummy value
--------------------------------
-1 ahoho=foo



1 afoo value=bar
3 abar value=dummy
5 adummy value=undef
```

Adding an extra -1-th element to the array all indices became values and all values became indices.

Associative arrays can have multiple indices just like normal arrays. For example:

```
sub printit(zz)
  print zz"foo","\n"
end sub
a"dummy","bar" = 13
a"dummy","foo" = 14
printit a"dummy"
print a"dummy","bar", _
  " ",a"dummy","foo"
```

will print

```
14
13 14
```

9.8.1 Notes for using associative arrays

Strictly speaking there are no associative arrays in ScriptBasic. Rather arrays can be used in associative mode. This implementation of associative arrays is simple and is powerful enough to use small arrays with a small amount of keys. The typical use of associative arrays is to get a feature like `record` in PASCAL or `struct` in language C. Note that this type of storage is less than optimal in case it is used for a huge number of keys and values.

Searching the keys in the array is linear. It means that accessing a single element needs time proportional to the size of the array. The external module "hash" provides an implementation, which is much more powerful and much faster for large associative arrays.

9.9 Using Array Mixed Mode

Once you will reach a point when you want to use an array in a mixed indexed mode. For example you want to handle an array of associative arrays. In other words you want to have an array, of which each element is an array itself.

For example you want to store the phone number, the room number and the name of the dog for each of your employee. You have 78 employees in your firm. In this case you need an array having 78 elements, each element being an associative array itself, with the keys "phone", "room", "dogname".

To do this you can

```
FOR I=1 to 78

  PRINT I,". ",EMPLOYEE[I]"phone"," ",EMPLOYEE[I]"room",_
        " ",EMPLOYEE[I]"dogname",\n"
NEXT I
```

You can mix the indices any order in any deepness. For example

```
ARR"1"[2,3][3]undef,"3",4"5"[66]
```

is correct. What you can use it for is another question though.

Note that

```
a[3][5]
```

is almost the same as

```
a[3,5]
```

but the latter is a bit more efficient (due to ScriptBasic internal algorithms). The details on differences can be read in the section See ⟨undefined⟩ [Reference Variables], page ⟨undefined⟩.

9.10 Expressions

To calculate values you will need expressions. The simplest expression is the one that contains only a constant number or string of the format described in the previous sections. A bit more complex expression contains variables, array elements, operators, function calls, and even parentheses.

ScriptBasic expression format is much the same as that of many other basic languages. The operators have precedence of the usual order, and you can alter the order of operator evaluation using parentheses. The operators are in order of precedence from the highest to the lowest:

`^` Power operator. `a^b` means `a` to the power `b`.

`*` Multiplication operator. This operator multiplies the operands.

`/` Division operator. This operator divides the left operand by the right operand and the result is usually a real number.

`\` Integer division operator. This operator divides the left operand by the right operand and the result is an integer.

`%` Modulus operator. This operator divides the left operand by the right operand and the result is the remainder after the division.

`+` Plus operator. This operator adds the operands and the result is the sum of the two operands.

`-` Minus operator. This operator subtracts the right operand from the left operand and the result is the signed difference of the two operands.

`=` Equality check operator. This operator checks if the operators are equal and results true if they are equal. Otherwise the result is false.

`<` Less than check operator. This operator checks if the first operand is less than the second operand and results true if yes. Otherwise it results false.

`<=` Less than or equal check operator. This operator checks if the first operand is less than the second operand or is equal to the second operator and results true if yes. Otherwise the result is false.

`>`Greater than check operator. This operator checks that the first operand is greater than the second one and it results true of yes. Otherwise it results false.

`>=` Greater than or equal check operator. This operator checks that the first operand is greater than the second operand or is equal to the second operand

`<>` Not equal check operator. This operator checks that the operands are NOT equal and it results true if they are not equal. Otherwise it results false.

`And` Logical and bit-wise AND operator. This operator calculates the logical and the bit-wise AND operation of the operands.

`Or` Logical and bit-wise OR operator. This operator calculates the logical and the bit-wise OR operation of the operands.

`Xor` Logical and bit-wise exclusive OR operator. This operator calculates the logical and the bit-wise exclusive OR operation of the operands.

`&` String concatenation operator. This operator concatenates the operands as strings.

`LIKE` Pattern matching operator.

These binary operators are evaluated from left to right. This means that 6-3+3 is 6 and not zero. This should be usual and obvious. 3 is subtracted from 6 and the last 3 is added to the result of the subtraction.

Operand types are not determined during compilation. There are no integer, real or string variables in ScriptBasic, any variable can hold any value (but only one at a time). Whenever a numeric operator gets a string value it converts the value automatically to numeric. The concatenation operator automatically converts the numeric operands to string.

Operators that work with both numeric and string operands do not convert the operands.

9.10.1 Operators

9.10.1.1 Power operator (^)

The power operator `a^b` calculates the b-th power of a.

The actual implementation of the power operator is quite complex. It tries to be as precise as possible. In case the calculation can be carried out using integer numbers only integer calculation is used. This way no unnecessary rounding errors are introduced.

If the mantissa is positive then the calculation is definite. However unlike other BASIC implementations ScriptBasic tries to calculate the result of the power operator even when the mantissa is negative. Actually power operator always results an integer or real number when possible. The implementation internally uses complex numbers and in case the imaginary part of the result is zero the real value is used as the result.

If either of the operands are `undef` then the result is `undef`.

9.10.1.2 Multiplication operator (*)

The multiplication operator multiplies the operands. If any of the operands are `undef`, the result is also `undef`. If any of the operands is real, then the result is real, otherwise it is integer. The string operands are converted to numbers.

9.10.1.3 Division operator (/)

This operator divides the first operand with the second. If any of the operands are `undef` then the result is `undef`. If the second operand is zero then the result is `undef`. If one of the operands is a real number then the calculation is carried out on real numbers and the result is real. If the operands are integer, but the calculation can not be performed to result an integer number without remainder then the operand values are converted to real. If both of the operands are integers and there is no remainder after the division the result is integer. String operands are automatically converted to numeric value. In case the option

```
OPTION RaiseMathError sbMathErrDiv
```

was used the operator will raise an error when the second argument is zero.

9.10.1.4 Integer division operator (\)

This operator divides the first operand with the second. If any of the operands are `undef` then the result is `undef`. If the second operand is zero then the result is `undef`. If one of the operands is a real number then the calculation is carried out on real numbers and the result is real. If the operands are integer then the calculation is performed using integer numbers and the result is truncated towards zero. String operands are automatically converted to numeric value. In case the option

```
OPTION RaiseMathError sbMathErrDiv
```

was used the operator will raise an error when the second argument is zero.

9.10.1.5 Modulus operator (%)

This operator calculates the remainder of the two operands. The operands are converted to integer value and the result is integer. If the second operand is zero the result is `undef`. In case the option

```
OPTION RaiseMathError sbMathErrDiv
```

was used the operator will raise an error when the second argument is zero.

9.10.1.6 Addition and subtraction operators (+, -)

These operators are both binary and unary operators. In their unary form they are used out of the precedence orders. Unary operators are always applied first, unless parentheses drive different calculation order.

If any of the operands is `undef` then the result is `undef`. If any of the operands is real the calculation is performed using real numbers and the result is real. If both operands are integer the calculation is performed using integer numbers and the result is integer.

9.10.1.7 Bit-wise and logical NOT (NOT)

This unary operator calculates the logical negate of the operand. The calculation is done on integer numbers, thus the operand is converted to integer value. The operator inverts each bit of the operand. If the operand is `undef` the result is -1, which is the internal value for TRUE.

Great care should be taken when using the NOT operator in logical expressions. The precedence of the operator is higher than that of any binary operator. Therefore the expression

```
not true or false
```

is true, while

```
not (true or true)
```

is false.

9.10.1.8 Equality operator (=)

This operator compares the operands. If both operators are `undef` the result is -1, which is the internal value for `TRUE`. If only one of the operands is `undef` then the result is zero, which is the internal value for FALSE. In other words `undef` is equal to `undef`, but nothing else.

If **any** of the operands is string then the comparison is done on strings. The non-string operand if any is converted to string. String comparison is case sensitive by default, but you can alter this behavior using the statement option. Two strings are equal iff they are the same length and contain the same valued bytes in the same order.

If the operands are numeric and at least one of them is a real number then the comparison is done on real numbers. Otherwise the comparison is done on integer numbers.

The result of the operator is always -1, which is the internal value for `TRUE` or 0, which is the internal value for `FALSE`.

To alter the behavior of case sensitive comparison you can use the statement

```
option compare sbCaseInsensitive
```

The comparing operator looks at the option "compare" and decides how to compare two strings. The global constant *sbCaseInsensitive* is pre-declared by ScriptBasic and can be

used anywhere in your program. To switch back to case sensitive comparison you have to issue the command

```
option compare sbCaseSensitive
```

The global constant *sbCaseSensitive* is also a pre-declared constant. Note that the option "compare" is also used by the pattern-matching operator like and also alters the behavior of directory listing under UNIX.

9.10.1.9 Not equal operator (<>)

This operator compares the operands. If both operators are `undef` the result is 0, which is the internal value for `FALSE`. If only one of the operands is `undef` then the result is -1, which is the internal value for `TRUE`. In other words `undef` is equal to `undef`, but nothing else.

If **any** of the operands is string then the comparison is done on strings. The non-string operand if any is converted to string. String comparison is case sensitive. Two strings are equal iff they are the same length and contain the same valued bytes in the same order.

If the operands are numeric and at least one of them is a real number then the comparison is done on real numbers. Otherwise the comparison is done on integer numbers.

The result of the operator is always -1, which is the internal value for `TRUE` or 0, which is the internal value for `FALSE`.

Later versions may change the behavior of this operator in case of string comparison allowing case-insensitive comparison.

9.10.1.10 Compare operators (<, <=, >, >=)

These operators compare numeric and string values. The result of the operation is `undef` if any of the operands is `undef`. If **any** of the operands is string the operands are converted to string value and string comparison is performed. If none of the operands is string but at least one of the operands is float value then the operands are converted to floating number and the comparison is done comparing floating numbers. Otherwise the comparison is done with integer numbers.

The result of the operator is always -1, which is the internal value for `TRUE` or 0, which is the internal value for `FALSE`.

9.10.1.11 Logical operators (and, or, xor)

These operators can be used for both logical and bit-wise operators. ScriptBasic does not have a separate type for logical values. Boolean values are stored in integer storage. The logical `TRUE` is represented as integer value -1 and the logical `FALSE` is 0. In other words the logical `TRUE` is an integer value, which is represented by a memory location of n bytes having all bits set to 1. On the other hand logical `FALSE` is an integer value represented by a memory location of n bytes having all bits reset.

The operators and, `or` and `xor` perform the calculation on integer values, If any of the operands is not integer it is converted to integer value before the operation takes place. The operations are performed on each bit of the operands.

This means that the operators can also be used for logical operations.

9.10.1.12 Concatenation operator (&)

This operator converts the operands to string value unless the operand is already a string value and results a string, which is the concatenation of the operands. The result string is placed in a newly allocated place, in other words the operation does not alter any of the operands.

9.10.1.13 ByVal operator

This operator is a very special one. This actually does nothing but results the operand unaltered. Why to have such an operator? The reason is to help calling functions and passing arguments by value. When an expression is passed as an actual value for a function argument it is passed by value. In other words the function gets the value of the expression. When a variable is used as the actual argument for the function the function gets the variable and not the value. Of course the function can use the value of the variable, but when it alters the variable, the original variable is also changed. To avoid this you can use the operator `ByVal` in front of the variable, which is passed to a function as argument. In this case the argument is not a variable anymore, but rather an expression and thus the called function or subroutine can not alter the value of the argument variable.

9.10.1.14 LIKE operator

The like operator is a pattern matching operator. Pattern matching is complex issue and is documented in a separate chapter in details. See chapter See ⟨undefined⟩ [Pattern matching], page ⟨undefined⟩.

9.10.1.15 Extension operators

Extension operators are defined and recognized by the ScriptBasic syntax analyzer but no execution is implemented behind. Thus using any of the extension operators will result an error "command is planned, but not implemented". Why to have these operators?

These operators can be implemented by external modules. An external module developer may decide to define the behavior of any of the operators. For example an extension may decide to implement a hashing algorithm and decide that `a->b` should result the actual hash value from hash `a` having the key `b`. Any program wanting to use this functionality can load the module using the statement `declare sub` and use the operator.

Operators that are undefined by default but can be used by external modules are:

```
?     !     #     ‘     @
+^    +<    +>    +?    +=    +*    +/    +%    +!    +#    +&    +\    +‘    +’
+    -^    -<    ->    -?    -=    -*    -/    -%    -!    -#    -&    -\    -‘
-’    -    ^^    ^<    ^>    ^?    ^=    ^*    ^/    ^%    ^!    ^#    ^&    ^\
```

```
^‘    ^’    ^    <^    <<    <?    <*    </    <%    <!    <#    <&    <\    <‘
<’    <     >^    ><    >>    >?    >*    >/    >%    >!    >#    >&    >\    >‘
>’    >     ?^    ?<    ?>    ??    ?=    ?*    ?/    ?%    ?!    ?#    ?&    ?\
¿     ?’    ?     =^    =<    =>    =?    ==    =*    =/    =%    =!    =#    =&
=\    =‘    =’    =     *^    *<    *>    *?    *=    **    */    *%    *!    *#
*&    *\    *‘    *’    *     /^    /<    />    /?    /=    /*    //    /%    /!
/#    /&    /\    /‘    /’    /     %^    %<    %>    %?    %=    %*    %/    %%
%!    %#    %&    %\    %‘    %’    %     !^    !<    !>    !?    !=    !*    !/
!%    !!    !#    !&    !\    ¡     !’    !     #^    #<    #>    #?    #=    #*
#/    #%    #!    ##    #&    #\    #‘    #’    #     &^    &<    &>    &?    &=
&*    &/    &%    &!    &#    &&    &\    &‘    &’    &     \^    \<    \>    \?

\=    \*    \/    \%    \!    \#    \&    \\    \‘    \’    \     ‘^    ‘<    ‘>
‘?    ‘=    ‘*    ‘/    ‘%    ‘!    ‘#    ‘&    ‘\    ‘‘    ‘’    ‘     ’^    ’<
’>    ’?    ’=    ’*    ’/    ’%    ’!    ’#    ’&    ’\    ’‘    ’’    ’
^
@<    @>    @?    @=    @*    @/    @%    @!    @#    @&    @\    @‘    @’    @@
```

The analyzer recognizes these operators as valid. They can be used as unary operator as well as binary operator. Used as binary operator they have the highest precedence, one level above the power operator and they are all evaluated from left to right.

Since build25 the unary operator `#` is implemented as identical operator resulting the argument of it unchanged. This is to aid the old BASIC programmers, who got used to write

```
open "file" for input as #1
```

9.10.1.16 Planned, Future Operators

Later versions of ScriptBasic will contain other operators implemented. The planned operators are:

`SHL` bit shift left

`SHR` bit shift right

9.11 Assignments (LET)

Assignment is the most important command of languages. This is used to assign calculated values of expressions to variables. You have already seen examples of it but we did not discuss it in details so far. An assignment is nothing else than a variable on the left side of an `=` character and an expression on the right side.

The command calculates the expression first. The expression may contain the variable that stands on the left side. In this case the variable holds its original value while calculating the expression. When the expression is fully evaluated the command releases the old value of the variable and then assigns the new value just calculated.

The calculated value can be string, integer, real number, even undefined or a whole array. The variable can be a global variable, local variable, argument of a subroutine or function or element of an array. For example:

```
' This assigns the value 18 to the variable A
A = 13 + 5
' This assignes the value "apple" to the array element
B[55] = "apple"
```

When the right hand side of the assignment is a whole array ScriptBasic creates a copy of the array and the left hand side variable will hold the new array.

Regarding evaluation order the assignment command first calculates the variable on the left side of the command and then it calculates the expression. Why is this important? There can be some special cases. Look at the following example:

```
function q
 z = z + 1
 q = z
end function

z =55
a[q()] = z
print a[56]
```

Will it print 55 or 56? Because the left side of the command is evaluated first it does print 56.

Some BASIC implementations allow a keyword `LET` to be used before the variable on the left side of the command before the variable. This is rarely used by programmer and is not allowed by ScriptBasic.

9.12 Operator Assignments

Most of the times we assign the value of an expression to a variable, which uses the variable itself. For example

```
A = A + 1
```

increments the variable *A*. To ease programming ScriptBasic allows the construct

```
A += expression
```

instead of

```
A = A + expression
```

This is a well known and widely used form by many languages, well readable, though not BASIC like. Likewise programmers can write

```
A -= expression instead of A = A - expression
A *= expression instead of A = A * expression
A /= expression instead of A = A / expression
A \= expression instead of A = A \ expression
A &= expression instead of A = A & expression
```

This is more readable for most of the programmers and results slightly faster execution for addition, subtraction, multiplication, division, integer division and string concatenation respectively.

9.13 Comments

You can insert comments into the source file. A line starting with the keyword `REM` or starting with the character ' (apostrophe) is treated as comment line.

```
REM This is a simple program that prints the words "HELLO" and "WORD"
REM to the screen
PRINT "HELLO WORLD"
```

Or you can write it

```
' This is a simple program that prints the words "HELLO" and "WORLD"
' to the screen
PRINT "HELLO WORLD"
```

However you can not use comments after commands, and therefore you can not write:

```
PRINT "HELLO WORD" 'This comment is invalid in ScriptBasic
```

or

```
PRINT "HELLO WORD" REM This comment is invalid in ScriptBasic
```

Note that the comment lines are tokenized and are "removed" from the source file after the lexical analyzer is done. Why is this important? Because multi-line strings may span more than one line. In this case the whole string that starts on a line being part of a comment will become comment. This way you can easily create multi-line comments that are quite useful for program documentation. For example:

```
' """

FILE:   test.sb
AUTHOR: Peter Verhas
DATE:   August 20, 2002

CONTENTS:
This is a program that contains nothing else but a multi-line
Comment.

"""
```

9.14 Including Files

You prepare your BASIC programs for ScriptBasic using some text editor. This is usually NOTEPAD under Windows operating systems and it is emacs or vi under UNIX. You enter the program line by line, but sometimes you may want to include other files into your program. One way is to use the copy and paste functionality of the editor. However this will lead unnecessary big text files and unmanageable projects. Instead you can use the `INCLUDE` command of ScriptBasic.

Whenever you want to include the file named '`myinclude.bas`' you should write

```
INCLUDE "myinclude.bas"
```

into your program. Although statements are not case sensitive, file names are on UNIX systems. On Windows systems the operating system preserves the case of the file names, but you can reference the file using any case letters.

If you want to maintain portability of your programs (and you usually should) use always lower case file names and reference them using lower case letters.

When you specify file names that reside in a different directory you can specify relative file names, or absolute file names including the path of the file. If '`myinclude.bas`' is in a subdirectory you should write:

```
INCLUDE "sudir/myinclude.bas"
```

Here we can mention another point of portability. UNIX uses the forward slash as file separator character. Windows command line interpreter use the back slash. In ScriptBasic you can use both separators on Windows and on UNIX as well. You can even mix them in a single path. We recommend that you use the forward slash character for the purpose.

If you use the back-slash characters in file names in ordinary BASIC strings like in the statement `OPEN` you usually have to double the character because this character plays the role of the escape character in strings. **This is not the case for the include statement.** You should write

```
REM a correct include statement
INCLUDE "subdir\myinclude.bas"
```

but

```
REM bad include statement
INCLUDE "subdir\\myinclude.bas"
```

will eventually fail to include the file.

Whenever you specify a relative directory, which is above the directory of the including file use the double dot notation of the parent directory, like

```
INCLUDE "../parentel.bas"
```

Whenever a relative file name is specified in an include statement the file name should be relative to the file that contains the include statement. For example you may have the three files:

```
C:\BASIC\INCLUDE\myinclude.bas

C:\BASIC\myprog.bas
C:\BASIC\INCLUDE\SYS\system.bas
```

The program '`myprog.bas`' includes the file '`myinclude.bas`' using the statement

```
INCLUDE "include/myinclude.bas"
```

The program '`myinclude.bas`' includes the file '`system.bas`' using the statement

```
INCLUDE "sys/system.bas"
```

The INCLUDE statement has another form:

```
INCLUDE mymodule.inc
```

This is similar to the format discussed above with the exception that there is no quote character around the file name. In this case ScriptBasic tries to locate the named file in one of the configured include directories. The file name in this case should not contain space.

Such include files usually belong to binary modules that ScriptBasic can dynamically load, and the included files declare the function implemented in the binary module.

The statement `INCLUDE` is processed before the variables, numbers, string and other lexical elements are recognized. This may result some strange behavior. For example the code:

```
T$ = """this is a multi-line string that
Include "otherfile.txt"
includes another file."""
```

is correct and results a string that contains the content of the file otherfile.txt. This may seem strange but it is the way ScriptBasic preprocessing handles the include directive.

Be aware of this feature as this may lead to strange errors when a multi-line string contains lines that start with the word `include`, `import` or `use`. Because nor the character `i` neither the character `u` has a special meaning back-slashed you can easily overcome the situation prepending a backslash `\` character before the special word. For example:

```
T$ = """this is a multi-line string that
\Include "otherfile.txt"
does not include another file."""
```

As your program goes larger and larger you split it up into included files. The code that includes them may include one file, including another file and so on. It may finally result some file included more than once. To avoid redefinition of thing and unnecessary code repetition ScriptBasic has another preprocessor statement named `IMPORT`.

`IMPORT` behaves the same way as `INCLUDE` does with the exception that it does not include the file if the file was already included by a previous `INCLUDE` or `IMPORT` preprocessor statement.

9.15 Internal preprocessors

ScriptBasic does not have a built-in macro preprocessor, like language C. However it has an open interface that third party developers can use to develop preprocessors. A ScriptBasic program may have one or more `USE` commands in it. This command should stand alone on a line being a preprocessor command.

The format of the command is:

```
USE preprocessor_name
```

After ScriptBasic has loaded the source code and has included all the files specified by the include statements it scans the lines containing a `USE` statement. When it finds one it loads the external preprocessor module compiled by the third party developer into a DLL or SO file and calls the preprocessor function. The preprocessors get total control over the source file loaded into memory and are free to alter the source code.

The preprocessor DLL or SO file should be specified in the configuration file. For example the line

```
preproc (
  internal (
    dbg "E:\\MyProjects\\sb\\Debug\\dbg.dll"
    )
```

```
...
```

from the default Win32 '`scriba.conf`' file defines what DLL file to load when the source file uses the preprocessor called `dbg` that actually happens to be the debugger preprocessor.

Internal preprocessors can also be loaded using the command line option '`-i`'. For example in case you want to start a ScriptBasic program from the command line using the command line debugger you can type:

```
# scriba -i dbg mybugous_program.bas
```

For more information on how to use a preprocessor see the documentation of the preprocessor.

9.16 Using external preprocessor

An external preprocessor is a stand-alone program that lets the programmer easily extend the language. The simplest available external preprocessor is the HEB preprocessor that lets a programmer embed ScriptBasic into HTML. This preprocessor is written in ScriptBasic and converts HTML embedded ScriptBasic code into pure ScriptBasic.

When ScriptBasic recognizes that an external preprocessor is to be started it does not read the source file but starts the external preprocessor and uses the file created by that. The external preprocessor should be an executable program that gets at least two arguments: the name of the file to preprocess and the file name to create; containing the preprocessed text.

The external preprocessor exits with the code zero in case it was executing successfully and should exit with the code 1 if an error occurred and the output file can not be used. This is the usual UNIX exit code convention for processes.

There can be more than one external preprocessors applied to a program. In case there are more than one preprocessors applied to a file then ScriptBasic starts each of them one after the other each getting the output file of the previous to work on.

The preprocessors should be configured in the configuration file before ScriptBasic can execute any of them. Each preprocessor should have a symbolic name. This is the choice of the person who configured the ScriptBasic installation. You can name a preprocessor as you like, it will not alter the behavior.

For example the HTML embedded basic preprocessor is named `heb` in the sample configuration file. To use this preprocessor you can use the option '`-p`' on the command line following with the name of the preprocessor:

```
# scriba -p heb myheb.bas
```

You can also assign file extensions to preprocessors. The sample configuration file contains the line:

```
epreproc$heb heb
```

This means that the extension `heb` is assigned to the preprocessor `heb`. Well, this is a stupid example because the extension is the same as the symbolic name of the preprocessor, but usually this is the convention. The extension stands after the `$` sign and the symbolic name of the preprocessor is the value of the configuration line. Thus

```
epreproc$extension preprocessor_symbolic_name
```

You can assign the same preprocessor to many extensions, for example:

```
epreproc$heb heb
epreproc$htb heb
epreproc$hb heb
```

can exist in the same configuration file. Whenever ScriptBasic processes a source file with the extension `hb` it will start the HEB external preprocessor.

A file may have multiple extensions. These are worked up from left to right. For example the command:

```
# scriba sample.heb.sql.bas
```

will start ScriptBasic starting first the preprocessor assigned to the extension `heb`, then the preprocessor assigned to the extension `sql` and finally the preprocessor assigned to the extension `bas`. If there is no preprocessor assigned to some of the extensions then they eventually will not be executed. This is not an error. (Note that there is no `sql` preprocessor currently for ScriptBasic, this is an artificial example.)

This behavior lets you to chain preprocessors.

External preprocessors have to be configured. You have to tell ScriptBasic what program to use for the specific preprocessing. To do this you have to use the configuration line:

```
eprep$exe$heb /usr/bin/scriba heber.bas
```

This configuration line tells ScriptBasic that the executable of the preprocessor having the symbolic name `heb` is `scriba heber.bas`. This is not actually the name of an executable. This is the start of the command line. The first element of it should be the name of the executable. The next elements are the command line options that precede the input and the output file name. Using this configuration line ScriptBasic will start another issue of ScriptBasic with the command line:

```
#/usr/bin/scriba heber.bas sample.heb.bas outputfile
```

where `sample.heb.bas` is the HTML embedded BASIC code we actually want to execute; '`outputfile`' is the file that the preprocessor has to create. When this process finishes ScriptBasic reads the generated output and executes the program.

Of course you can use any other executable as preprocessors. You can write preprocessors in C, C++, Perl or in any other language that can be started on the command line, is capable reading and writing file and is able to signal error via exit code. Note that ScriptBasic exists with the last non caught error code.

Before ScriptBasic is able to start the external preprocessor it has to calculate the name of the output file that the preprocessor has to write. Each preprocessor has to be configured to use a temporary directory to store the preprocessed file. To specify the directory you have to use the configuration line:

```
eprep$dir$heb /etc/temp/heb/
```

This means that the temporary directory for the files created by the preprocessor with the symbolic name "heb" is the directory '`/etc/temp/heb/`'. When a file is to be preprocessed with this preprocessor ScriptBasic calculates a unique file name based on the name of the source file and asks the external preprocessor to put the result into that file in this directory. Finally the command line that ScriptBasic starts for the example above is:

```
#/usr/bin/scriba heber.bas sample.heb.bas /etc/temp/heb/EACAPAOALAEAHAHAAANAJAOALAG
```

The algorithm used to calculate the unique file name is the same as for the cache file name.

9.17 #! /usr/bin/scriba

Your basic program may contain a first line that looks like the title of this section. This may seem strange for Windows users, but this is usual in UNIX. On UNIX systems such a first line tells the operating system, which program to use to execute the program. This is usually `/usr/bin/scriba` on UNIX systems, but it is not a must The ScriptBasic interpreter may be in a different location or may have different name.

When the interpreter starts the program it sees this first line and ignores it. To ease life of ScriptBasic users this kind of first line is ignored on Windows systems.

9.18 @goto start

Your basic program may contain a first line that looks like the title of this section. This may seem strange for UNIX users, but this is usual under Windows. On Windows systems such a first line is used to transfer the execution of the Windows command interpreter to the end of the command file, where the interpreter is invoked for the same file.

This looks like:

```
@goto start

   scriptbasic program lines

rem """
:start
@echo off
scriba %0 %1 %2 %3 %4 %5 %6 %7 %8 %9
rem """
```

assuming that `scriba` is in the path this file named something `xxxx.cmd` can be started from the command line and will execute the BASIC program. The first line is ignored by the ScriptBasic interpreter, but the Windows command processor will jump to the line containing `:start`. On this line the ScriptBasic interpreter is started and reads and interprets the lines. The first line is ignored by ScriptBasic, the last few lines are comment at least for ScriptBasic.

To ease portability such a first line is ignored when executing a BASIC program under UNIX.

9.19 #!/@goto

After the previous two sections here is a small trick that allows you to have a ScriptBasic program that can be executed under UNIX as well as under Windows just typing the name of the BASIC program file and without any modification.

```
#! /usr/bin/scriba
goto start
start:

scriptbasic program lines

rem """
:start
@echo off
scriba %0 %1 %2 %3 %4 %5 %6 %7 %8 %9
rem """
```

Unfortunately under Windows NT the command interpreter will complain about the very first line. However in addition to that there is no other issue.

10 Interacting with the user

Although this topic is detailed elsewhere we present a short introduction here, because of its importance. Interacting with the user is the most important action a program takes. There are many ways to do it, but the most frequently used is printing to the screen and reading from the keyboard and interpreting command line arguments. To do this ScriptBasic has the command print and the command line input.

10.1 Print

The print statement can either print to a file or to the standard output, which is usually the terminal window. If no file number is specified after the keyword the statement print prints to the screen. It takes a list of expressions, formats them and prints the values to the screen.

```
print "haho!"
print "hahaho!"
print
print "kukac\n"
print "oooh"
print "The number is ",number
```

The list of expression is one or more expressions separated by commas. Old implementations of the BASIC language used `;` for that and inserted a space between the printed item. These old BASIC languages used the comma as list separator to put the next printed item on the next tab position. There is nothing like that in ScriptBasic. The comma simply separates the list elements, which are printed adjacent. Comma does not insert space between them and does not position on tab. `;` as list separator is not allowed.

In case you want to position something in tab position you can

```
print "hello\ttab"
```

use the `\t` escaped character.

10.2 Input

ScriptBasic has the usual input statement, however a slightly different from other BASIC implementations. Input statement in other basic languages reads the keyboard and puts a string, real or integer number into the variable depending on the type of the variable. In ScriptBasic there is no type of the variables, any variable can hold any type of value. Thus the input statement can not know if the input is to be read as integer, real or character string as typed.

Because of this ScriptBasic reads the characters as typed into the variable on the input statement as string. Because this is the usual behavior of the BASIC languages for the command `line input` ScriptBasic also names this statement `LINE INPUT`.

This command reads a whole line from the file specified after the # sign or from the standard input if no file number was specified. The string read is put into the variables listed on the command. Each variable gets a line including the line terminating new line character.

```
print "Give me a line:"
line input a
print "This is a=",a,"This was a\n"
```

To remove the new line from the end of the read string you can use the function `chomp`.

10.3 Handling command line arguments

Though the command line is parsed by the interpreter up to the name of the executable basic program name and all other command line options and arguments are passed to the basic program to handle. To access this part of the command line the basic program should use the function `command()`. This function does not need any argument and returns a string containing the command line arguments.

In the variation STANDARD the command line returned by the function `command()` contains the parameters separated by a single space. Even if the real command line contains multiple spaces or tabs between the parameters it is converted to a list of parameters separated by single space.

```
' This program demonstrates the command function
' start the program using the command line:
'     scriba commandline.bas a b c d e f
'
print "The command line was:\n",command(),"\n"
```

will print

```
The command line was:
a b c d e f
```

11 Name spaces

ScriptBasic does not have real name spaces to separate modules from each other. There are no such things as public and private variables of modules or classes. Whenever you develop a module you have to trust the programmer using the module that he or she will use the module in the way it is intended and the way you hopefully documented. A programmer should not use the internal variables of a module not because he can not but because he is supposed not to.

ScriptBasic name spaces only help the programmers to avoid accidental name collisions. When you develop a module you start it saying

```
MODULE MyModule
```

After this line until the end of the module which is noted by the line

```
END MODULE
```

will belong to the specific module and all global variables will belong to that module.

However all these module and name space handling is done on low level altering the names of the variables and functions. When you start to write a program and you write:

```
A = 3
print A
```

you actually use the variable *main::A*. This is because the default name space is `main`, and if there is no name space defined in a variable its name is altered so that the variable will belong to the current name space. The default name space is called `main`. Try the following:

```
A = 3
print main::A
```

It prints 3. Surprising? Try the following:

```
A=5
module boo
A=3
print main::A
end module
```

It will print 5. This is because *main::A* is 5, *boo::A* is 3 and the variable name *main::A* is not converted to *boo::main::A*, because it is explicitly denoted to belong to the name space `main`.

Name spaces can be nested. You can write:

```
A=1
module boo
 A= 2
 module boo::baa
 A= 3
 print A
 print boo::A
 print main::A
end module
end module
```

which will print 321.

To ease the nesting of modules you can write the same code as

```
A=1
module boo
 A= 2
 module ::baa
 A= 3
 print A
 print _::A
 print main::A
end module
end module
```

When the module name in the module declaration starts with double colon ScriptBasic knows that the module is to be nested into the current module. The variable name _::A means: the variable A of the surrounding name space. This is the same as the operating system directories. You can think of name spaces as directories and variables as files. Whenever you write

```
../../mydir/file.c
```

a similar construct may say

```
_::_::mynamespace::variable
```

When the parser sees the end module statement it always returns to the previous name space. You can start and close modules many times, ScriptBasic will not complain that the module was already defined. You can even totally neglect the module statement and you can write the above program as

```
main::A=1
boo::A= 2

boo::baa::A= 3
print boo::baa::A
print boo::A
print main::A
```

This is a bit less readable. Name spaces can serve another purpose. See the following code:

```
string = "AF"
hex = 0
while string <> ""
 chr = mid(string,1,1)
 string = mid(string,2)
 hex = hex*16 + asc(chr) - asc("A")
wend
print hex
```

when you try to compile you will probably get many errors. Why? Because string, `chr` and `hex` are predefined functions and can not be used as variable names. What to do then? You can use them as variable names when you specify that they are variables:

```
::string = "AF"
::hex = 0
while ::string <> ""
 ::chr = mid(::string,1,1)
 ::string = mid(::string,2)
 ::hex = ::hex*16 + asc(::chr) - asc("A")
wend
print ::hex
```

When you write var::string, *::chr* or *::hex* ScriptBasic will know that they are variables belonging to the current name space. This is a bit weird, so you better avoid using function names and predefined function names as variable names.

12 File Handling

Tis chapter discussed how ScriptBasic handles files, how you can open, close, read, write, delete, create, rename, copy, move files.

12.1 Opening and creating files

ScriptBasic handles the files the same way as any other BASIC type language. You open a file, read from it, write to it and finally close the file. To make a jumpstart see a simple example:

```
open "myfile.txt" for output as 1
print#1,"This is the first line of myfile\n"
close 1

open "myfile.txt" for input as 1
line input #1, a
close 1
print a
```

This simple program opens the file named '`myfile.txt`' in the current directory, prints a single line into it, and closes the file. Next time it opens the file for reading, reads a line from it, closes the file and prints the line read from the file to the screen.

When you open a file you have to have a file number that you want the file associated with. In the example above this number is 1. This is called many times the "file number". Whenever you do something with an opened file you have to use the file number.

There are more things than reading from a file and writing to a file. You can read from a certain position on a file and you can write to a certain position. You can determine the length of a file and dates and times the file has, like last time it was accessed, modified or created.

12.2 Text and binary files

Before going into details we have to talk about the different types of files and the different ways a file can be opened.

A file is a byte stream for ScriptBasic. You can open a file for reading, writing, or to do both of these operations without closing and reopening the file. Some operating systems, like Windows NT, distinguish between text files and binary files. Other operating systems, like UNIX do not make such a strict distinction.

Text files usually contain lines of text. Binary files may contain any code. When you open a file that you wan to write textual data into, or you know that the file contains textual data, you have to use one of the text mode opening. When you want to handle arbitrary data, use binary type of opening.

Calling a file to be text or binary under Windows NT is not precise. All files are the same. The differentiation between textual and binary should be set up on the way we use

the files. You can open a text file in binary mode and you can open a binary file in text mode. A file itself is nor binary neither text. The difference is how we handle them.

Binary files are simple. Bytes follow each other and when we read or write such a file the operating system reads or writes the bytes as they are without any conversion. Whenever you want to go to a certain position of a file using the command `seek`, you can without problem.

Textual files are a bit different. Whenever you open a file in textual mode and start to read it, the operating system converts each carriage-return and line-feed character pairs to a single line feed character. Whenever you write to a file opened in textual mode all line-feed characters produce two characters in the output file: a carriage-return and a line-feed character.

Setting the position in a textual file is also tricky. Should we count the bytes in the file or the bytes we have wrote to the file. They are different. If we have send x pieces of new-line characters to the file that generated two times x bytes. The safest rule is: *do not position within a text file.* Read it from start, write it from start or append to it. The not so safe rule is position to "non-calculated" position. Your calculation may be erroneous, but if you position to a value that was reported by the function `pos()` beforehand, you may be safe.

UNIX operating systems handle all files in one way that makes life simpler. Text file lines are separated with a single line-feed character and opening a file in textual mode is just the same as opening in binary mode. If you write a program for UNIX you need not worry about textual and binary mode. But this is true only so long as long your program is running under UNIX. Whenever someone starts to use your program under Windows he or she may start to report you bugs that did not appear under UNIX. Therefore it is recommended that you use the appropriate file opening modes under UNIX the same way you would under Windows NT.

There are five different ways you can open a file. These are

```
Input
Output
Append
Random
Binary
```

There is an extra mode called `socket` that opens connection to a remote machine and not a file. That will be detailed later.

The first four modes are textual modes. Open a file for input whenever you want to read data from the file. If you open a file for output you can write to the file. However all data that was stored in the file is destroyed, and in case the file did not exist it is created. If you want to write new data to a file still keeping the existing content open the file for appending to it, using the keyword `append`.

`Random` gives a relative freedom to open a file for both reading and writing. When you open a file using the keyword random, you can both read from the file and write to the file. You can, for example, read from the file until you find the end of the file and then append after the current content new lines of text. In another scenario you can read the current content, seek to the position zero, which means the start of the file and rewrite the existing

content. However you should only use the mode random only when you really want to write the file. Operating system permission may allow you to read a certain file and not to write. Trying to open the file in mode random will not be successful in that situation and prevent your program's ability to read the file content.

The last mode is `binary`. This opens the file for both reading and writing in binary mode. You can send any data to the file without conversion and you can position the file pointer to any position without worry.

All file modes but `input` create the file for you if it did not exists and all modes but `input` create the directory for the file if it did not exist. In other words there is no need to ensure that the directory exists for a file before opening it using any of the outputting modes.

The format of the open statement is:

```
open file for mode as [ # ] file-number [LEN=record_length]
```

The parts between the `[` and `]` characters are optional.

The '`file`' is the name of the file to be opened. The mode is one of the keywords above. The `file-number` is the number you want to refer to the opened file. This number should be between 1 and 512 in the current implementation of ScriptBasic. 512 is the maximum number of concurrently open files in ScriptBasic, unless the underlying operating system gives a lower limit. The file number should be free to use, in other words no two files can be opened at a time with the same file number. However if you close a file, the file number is released and can be used in subsequent open statements.

The `#` character is optionally allowed before the file number for compatibility reasons with other BASIC interpreters.

The last optional part specifies the record length in terms of bytes. If this record length is not defined the file is treated as series of bytes. If the record length is larger than one the positioning statements count the position and length in records instead of bytes.

12.3 Switching between binary and text mode

binary mode files

Whenever you have opened a file in a mode you may want to switch the mode. You can not switch between reading and writing mode, but you can switch between binary and textual mode. To do this there are two commands:

```
binmode [#] fn
```

and

```
textmode [#] fn
```

The sign `#` is optional. The argument *fn* should be the file number of an opened file. These commands are implemented on the Windows operating system, but can also be used under UNIX with no effect. The use of these commands under UNIX results more portable code.

There are two files automatically opened in almost any environment. These are the standard input and the standard output. You can switch the mode how the program handles these files using these commands. You can say

```
Binmode input
Binmode output
```

to switch the standard input and output to binary mode and also

```
Textmode input
Textmode output
```

to switch the standard input and output to text mode. Use of these commands is extremely useful writing Windows NT cgi programs that output binary data, for example a png file.

12.4 Getting a free file number

At certain situation you do not know a number which is sure available to be associated with a file. In that case you can use the function `FreeFile()` to get a file number usable in the next open statement.

The following code demonstrates the usage of the function `FREEFILE`:

```
fn = FREEFILE
Open "myfile.txt" for input as fn
```

The first line of the code stores a number in the variable fn, which can be used in the next line as a file number. You can print out the actual value, store it in one or more variables.

To be absolutely safe you can also check if there is any available file number. If there is no available file number then the function `FREEFILE` returns `undef`.

```
fn = FREEFILE
if IsDefined(fn) then


  open "myfile.txt" for input as fn
else

  ' do whatever you have to do

  ' when you can not open the file
end if
```

There is another way to automatically get the first free file number in an `OPEN` statement.

In case the file number is specified using a single variable initialized to zero the interpreter will find automatically the first available file number and will put this value into the variable. This way you can also write:

```
fn = 0
ON ERROR GOTO ERROR$FILE$NOT$OPENED
open "myfile.txt" for input as fn

' do whatever you want with the file

STOP
```

```
ERROR$FILE$NOT$OPENED:
  ' do whatever you have to do
  ' when you can not open the file
```

12.5 Positioning in a file

When you open a file for `RANDOM` or `BINARY` access you may want to move inside the file back and forth before reading or writing data. There are many functions and statements that help you to move the file pointer. Before going into details, let's explain what the file pointer is.

A file is a series of bytes stored on the disk. Whenever you read a byte from a file you do it from a certain position. The file pointer of the opened file identifies this position. When you read a line or some bytes from a file the file pointer automatically moves after the last character of the line. Therefore the next read will go on reading the next available unread character of the file. The same is true for writing the file. Whenever you write bytes to a file the file pointer associated with the opened file will move after the last position written.

If the file pointer is positioned after the very last byte of the file reading will result an end-of-file signal and the function `EOF(fn)` will be `TRUE`. If we try to write to the file in the same situation the length of the file will increase,

To get the actual length of an opened file you have to call the function `LOF(fn)` that stands *for Length Of File*. To get the current position of the file pointer you have to call the function `POS(fn)`. To position to a certain position of an opened file you have to write:

```
seek #fn, position
```

where position is the number of bytes, or records from the start of the file. When you want to position to the start of the file, you can therefore write either

```
seek #fn, 0
```

or

```
Rewind #fn
```

`Rewind` is nothing else than a short form for `seek#fn,0` because it is often needed and is more readable.

The functions `LOF` and `POS` return values in terms of records. If there is no record length specified in the statement open you can treat the return values as counts of bytes. On the other hand if there is a record length larger than one specified on the statement open these functions return a value in terms of records. `LOF` tells you how many records there are in the file and `POS` tells you which record the file pointer is positioned in.

You have to be careful interpreting the returned values of these functions with files opened with record length more than one. `LOF` tells you how many records there are in the file actually, but there may be some more bytes after the last complete records. `POS` tells you which record the file pointer is positioned in, but it does not guarantee that the file pointer is positioned on the first byte of the record.

There is a function named `FILELEN` that gives the length of a file based on the name of the file. The argument for this function is the name of the file, while the argument of `LOF` is the file number of the opened file.

Another difference is that `FILELEN` returns the size of the file in terms of bytes always and never in terms of records. This behavior becomes obvious if you recall that record length is specified when a file is opened and `FILELEN` works on unopened files.

12.6 Reading and writing files

PRINT #fn

To write into a file you can use the statement print in the form:

```
PRINT#fn, expression list
```

This is the same format as the ordinary print statement with the exception that a file number is specified that references an opened file. The expression list is printed into the file. If you want to start a new line in the file you can do it in one of two ways. You can write:

```
PRINT#fn, "\n"
```

or

```
PRINTNL#fn
```

In the first case the string containing a new line character is printed to the file. In the second case we wrote a print-new-line command. Note that you can print a new line character to the screen in three ways:

```
PRINTNL
PRINT
PRINT "\n"
```

The second format can not be used to print a new line into a file, the command

```
PRINT#fn
```

is syntactically incorrect. This is a restriction caused by the parser that ScriptBasic is built on. Later versions may allow this format.

To read from a file you have a statement and a function. The statement

```
LINE INPUT #fn, variable
```

will read a line from the file. The length of the line is not limited except that there should be enough memory. This statement will result a string in the variable that contains the line including the line terminating new line unless the line is terminated by the end of the file without closing new-line character.

For binary files that are not composed of lines the function `input()` should be used. This function reads from a file the given number of characters or bytes and results a string that contains the bytes read. It has the format:

```
Variable = input( NumberOfBytes, fn)
```

The number of bytes actually read and put into the variable by the function can be determined using the string function `LEN()`. If there are less number of bytes or records

in the file from the actual file pointer until the end of the file than the required number of bytes the function results a shorter string containing the available bytes.

12.7 Getting and setting the current working directory

When a program runs under the UNIX or the Windows NT operating system there is a directory where the program is started. This directory is not necessarily the directory where the executable code is or where the basic program is. Sloppy saying: this is the directory where the user prompt was when the user typed in the command line starting the program. This is called the *current working directory*.

Whenever a file name is specified it can either be an absolute file name or a relative file name. Relative file names are always relative to the current working directory. For example the current working directory is '`/home/pvc/scriba/example`' and the file name is '`hello.bas`' then the operating system know that the actual file we refer to is '`/home/pvc/scriba/example/hello.bas`'. If we refer to the file '`../source/getopt.c`' then the operating system considers the file '`/home/pvc/scriba/source/getopt.c`'. This is because the '`..`' directory means : one directory above. This is the same in Windows NT as well as in any UNIX variant. (OpenVMS is a bit different.)

Using relative file names is easier than using the full path to each and any file name. ScriptBasic provides a command and a function to handle the working directory. The function `curdir` has no arguments and returns a string containing the current working directory path. The command `chdir` changes the current working directory according to its argument. For example:

```
print curdir,"\n"
chdir ".."
print curdir,"\n"
```

will print

```
/home/pvc/scriba/examples
/home/pvc/scriba
```

As argument to `chdir` you can define any existing directory name as absolute path or relative to the current working directory. If the directory does not exist or there is some other condition that prevents the change of the current directory an error occurs.

Note that this command is disabled in the Eszter SB Application Engine variation of the interpreter and should be disabled in all multi-thread versions. The reason is that the current working directory is set for all interpreters running in the same process using this command and may generate failures in other scripts.

12.8 Locking a file

So far we did not consider the case when more than one programs are trying to read or to write a file. In real life, especially if you write CGI programs this can happen very often. Assume that you want to count the number of visitors. The CGI program fragment making the counting can be something like this:

```
1 Open "counter.txt" for input as 1
2 Line input #1,Counter
3 Close #1
4 Counter = Counter +1
5 Open "counter.txt" for output as 1
6 Print #1, Counter
7 Close #1
```

At first look there is no problem with this. But at second thought there is. There can be many CGI processes that run parallel. There is no guarantee that one program finishes its working before the next one starts. Processes are run parallel, and it means that the operating system runs a little piece of one process then runs another. Let's imagine the following unfortunate, but likely situation.

A CGI process runs the lines 1 to 3 and reads the counter value, let's say 100. Then the operating system hangs the running of this process for a while and runs another CGI process again the lines 1 to 3. This process also reads the counter value which was still not changed and is still 100. Then the first process gets focus again and it writes the incremented value 101. The second process, whenever it gets its time slice to run writes also the value 101 to the file. Both hits counted from 100 to 101 instead of one counting from 100 to 101 and the other from 101 to 102.

What is the solution?

The solution is to lock the file. A file can be locked when it is opened. We open the file for random access, lock it, read from it, increment, write the incremented value and close the file. No one else can intercept the file access because the file is locked. How does it look like in ScriptBasic?

```
1 Open "counter.txt" for random as 1
2 lock#1,write
2 Line input #1,Counter
3 Counter = Counter +1
4 rewind#1



5 Print #1, Counter
6 Close #1
```

The command `lock` does lock the opened file. (Obvious?) The first argument is the file number of the opened file. The second argument is the locking type. This type currently can be `write`, `read` and `release`. The meaning of the different types are:

`Write` The file is locked for writing. Only the locking process can access the file until the lock is released.

`Read` The file is locked for reading. Anyone can read the file but no one can write it until the lock is released.

`Release` The file lock is released.

If the file is already locked by another process so that the actual locking cannot be performed the instruction `lock` waits until the other process releases the file.

The ScriptBasic instruction `lock` is an advisory lock. It means that it works only if all programs wanting to access the file use the lock statement before reading or writing the

file. If a process locks a file and another process just goes on reading or writing the locking method may not work. The actual behavior is dependent on the operating system and as such you have to treat lock working more than advisory as non-guaranteed, or undefined behavior.

Actually the `flock` system call is used to implement the lock statement of ScriptBasic. This type of lock does not prohibit any process to read or write a locked file on UNIX. On Window NT operating system `flock` does not exist. The file locking functionality is implemented using the `LockFileEx` system call and locking the first 64Kbyte of the file. This results write and read failure in a process that tries to read or write the first 64Kbyte of the file and UNIX like behavior if it tries to access the file above the 64Kbyte.

12.9 Locking file range

ScriptBasic provides another type of file locking. This lock locks a range of a file. You can do it using the statement:

```
LOCK RANGE #fn FROM start_pos TO end_pos FOR mode
```

fn is the opened file, *start_pos* and *end_pos* are expressions resulting integer values meaning the first and the last position of the locking range in terms of records. The parameter *mode* is the same as for the file locking mechanism, it can be *read*, *write* or *release*.

Note that the positioning uses the same zero offset numbering as the instruction `seek`. In other words the first record of a file is position 0.

See the following example:

```
REM locktest1.bas
open "locktest.txt" for output as 1
lock region#1 from 1 to 5 for write

for i=1 to 5
print #1, "A"
next I
print "5 bytes are done\n"

for i=1 to 5
print #1, "B"
next I
print "10 bytes are done first 5 bytes are locked\n"
line input a
close 1
```

If you run this code in two different terminal windows one will lock the file range and the other will go on and wait for user input. When you press the key ENTER to go on the program grabbing the lock closes the file and the program in the other terminal windows writes out the text and starts to wait. On the other hand if you start '`locktest1.bas`' in one terminal window and when it starts to wait for user input you start the following '`locktest2.bas`' in another window, they do not interfere, because they lock different regions of the file.

```
REM locktest2.bas
open "locktest.txt" for output as 1
lock region#1 from 6 to 10 for write

seek#1,6
for i=1 to 5
print #1, "D"
next I
print "done\n"
close 1
```

This locking method implements advisory locking, and the behavior is the same as in the case of file locking. The interpreter calls the system function `fcntl` on UNIX systems, and `LockFileEx` on Windows NT. This results different behavior when programs try to read or write a file region locked by another process.

In other words all programs should behave and lock the file or the region before accessing it.

12.10 Truncating a file

When you write to a file it is automatically extended by the operating system and grows. On the other hand files do not shrink automatically. When you write into file before the position of the end of the file the bytes after the written region are still valid and remain in the file. The exception is when you open a file for `output`. In that case the original file is deleted and you can start to write of length zero,

However you may want to open a binary file, read from it, write into it and sometimes you may want to decrease the size of the file. The instruction truncate does it for you. The syntax of the instruction is:

```
truncate#fn,size
```

Where *fn* is the opened file number, and size is the new size of the file in terms of records. The content of the file after the position `size-1` is lost. Note that the argument to the instruction truncate is the desired length of the file, which is the position of the last byte or record plus one. Positions start with zero offset just like in file positioning statements.

If the specified length is larger than the actual length of the file then the file is elongated with bytes containing the value zero padded to the end of the file.

12.11 Deleting a file or directory

Using the command truncate you can truncate a file to size zero, but the empty file is still there. To totally delete a file you have to use the command `DELETE`. The syntax of the command is

```
delete expression
```

where the expression is the name of the file or directory to be deleted. When issuing this command to delete a directory the directory should be empty otherwise the command fails and an error occurs. There is another command with the syntax

```
deltree expression
```

that deletes a directory even if the directory is not empty. Note that this command is very dangerous, because it forcefully deletes full directory trees and therefore you have to use this format of the command with exceptional great care.

12.12 Creating a directory

Although the `OPEN` statement automatically creates the directory for the file whenever you want to create a new file you may want to create a directory without creating a file in it. For the purpose the statement `mkdir` can be used. The syntax of the command is very simple:

```
mkdir expression
```

The expression should result a string, which is the directory to be created. The command recursively creates the directories that do not exist by the time the command is executed and need creation in order to create the final directory. For example you can issue the command

```
mkdir "/usr/bin/scriba"
```

without issuing the commands

```
mkdir "/usr"
mkdir "/usr/bin"
```

The directories '`/usr`' and '`/usr/bin`' are automatically created in the unlikely case they did not exist.

The command executes without error if the directory is created or if the directory already existed. If the directory can not be created an error occurs. The reason for directory creation failure can be numerous. The most typical is access control prohibiting for the process directory creation. Another usual problem is when some sub-path of the desired directory already exists, but is a file. For example we want to create the directory /usr/bin/scriba/exe and /usr/bin/scriba already exists and is a plain file.

12.13 Setting file parameters

Files usually have parameters in addition to the information contained in the file itself. Such information is the owner of the file, the group the file belongs to, creation time, modify time, access time, change time and so on. To set these parameters of a file ScriptBasic provides the command `set file`.

The syntax of the command is

```
set file filename parameter=value
```

where '`filename`' is the name of the file for which the parameter is to be set, the `parameter` is the name of the parameter to change and value is the desired value of the parameter.

The `parameter` can be

`Owner` to set the owner of the file. This owner is a user of the system identified by its login name. The owner of a file is usually the person who ran the program that created the file. The super user or administrator can change the owner of a file passing the ownership to any other legal user. The use of the command can be different on Windows NT from UNIX. Windows NT uses different NT domains and user names can be noted in the form `DOMAIN\USER`. The owner of a file on a machine belonging to one domain can be a user from another domain (that the domain the machine belongs to hopefully trusts). If the user name specified in the command `chown` contains a `\` character (do not forget to use `\\` in the strings) the user name is looked up in the domain specified in the name. If the name does not contain any domain specification the local user database is used.

`CreateTime` to set the creation time of a file. The value for the time should be expressed in terms of seconds since January 1, 1970. 00:00. This is the usual measure of time on UNIX systems. This parameter does not exist on UNIX operation systems. UNIX maintains only modification, access time and change time, therefore setting `CreateTime` is not possible under UNIX and trying to perform the set file command for this parameter results error with error code `sbErrorNotimp`.

`ModifyTime` to set the last modification time of a file. The value for the time should be expressed in terms of seconds since January 1, 1970. 00:00. This is the usual measure of time on UNIX systems.

`AccessTime` to set the last access time of a file. The value for the time should be expressed in terms of seconds since January 1, 1970. 00:00. This is the usual measure of time on UNIX systems.

If the file does not exist or some other condition prevents the successful change of the parameter the program generates an error with one of the error codes `sbErrorSetCreateTime`, `sbErrorSetModifyTime`, `sbErrorSetAccessTime`.

Note that Windows NT file system (NTFS) and UNIX operating systems store file time values in GMT. Do not forget to convert the local time to GMT if needed when changing some of the file time values. Other file systems, like FAT may store the file times as local time. Be careful.

12.14 Listing files

Up to know we have discussed instructions and operations that were dealing with files. To access a file you needed to know the name of the file. Many times the program does not know the file and needs to get the list of all files in a certain directory. To do this you can use the commands `OPEN DIRECTORY`, `CLOSE DIRECTORY` and the function `NextFile`.

The command `OPEN DIRECTORY` opens a directory for getting the list of the file. The command reads the list of the files and creates an internal list of the files.

The function `NextFile` can be used to retrieve the files of the directory opened one by one. When the last file name has been retrieved or when there is no need to retrieve the remaining file names the command `CLOSE DIRECTORY` should be used to close the listing.

12.14.1 Open directory

The syntax of the command open directory is

```
OPEN DIRECTORY dir_name PATTERN pattern_value OPTION option_value AS dir_number
```

The parameter `dir_name` should be an expression evaluating to a string, which is the name of the directory for which the listing is needed. It can be absolute path or relative to the current working directory. It may but need not contain the trailing slash.

The parameter `pattern_value` should be an expression evaluating to a string. When the list of the files is gathered this pattern is used to select the files. Only the files matching the pattern will be included in the list of files. The pattern matching uses the same algorithm as the operator `LIKE`. Altering the joker and wild characters for the operator `LIKE` also alters the pattern matching mechanism of file selection. The command `OPEN DIRECTORY` leaves the pattern matching memory in a non-matched state. In other words the function `JOKER` will return the `undef` value after executing an `OPEN DIRECTORY` command until the next `LIKE` operator is executed.

Pattern matching and selection is done during the execution of the command `OPEN DIRECTORY` the other file list handling commands and the function `NextFile` do not interfere with the pattern matching.

Pattern matching while building up the list is a time and processor consuming tasks. If you do not want to perform pattern matching just getting all the file names use value `undef` or empty string as pattern value. In either case pattern matching is skipped during file list build up. Using `*` to get all files is a bad idea unless your aim to make your CPU generate heat.

Note that pattern matching can either be case sensitive or insensitive based on the setting of the option '`compare`'. This option alters the behavior of pattern matching during file list build up on UNIX systems, but not under Windows. Under Windows NT pattern matching during file list build up is always case insensitive, no matter how the option '`compare`' is set.

The parameter `option_value` should be an expression evaluating to an integer value. This value is used to drive the behavior of the command. Each bit of the value has a special meaning. To set the correct value ScriptBasic predefines several global constants that you can use. These constants can be used in this value. If more than one constant is needed you have to use the operator `and` to connect them. The predefined constants and their corresponding meanings are:

`SbCollectDirectories` Collect the directory names as well as file names into the file list.

`SbCollectDots` Collect the virtual . and .. directory names into the list.

`SbCollectRecursively` Collect the files from the directory and from all the directories below.

`SbCollectFullPath` The list will contain the full path to the file names. This means that the file names returned by the function `NextFile` will contain the directory path specified in the open directory statement and therefore can be used as argument to file handling commands and functions.

`SbCollectFiles` Collect the files. This is the default behavior.

`SbSortBySize` The files will be sorted by file size.

`SbSortByCreateTime` The files will be sorted by creation time.

`SbSortByAccessTime` The files will be sorted by access time.

`SbSortByModifyTime` The files will be sorted by modify time.

`SbSortByName` The files will be sorted by name. The name used for sorting will be the bare file name without any path even if the option bit `SbCollectPath` is specified.

`SbSortByPath` The files will be sorted by name including the path. The path is the relative to the directory, which is currently opened. This sorting option is different from the value `sbSortByName` only when the value `sbCollectRecursively` is also used.

`SbSortAscending` Sort the file names in ascending order. This is the default behavior.

`SbSortDescending` Sort the file names in descending order.

`SbSortByNone` Do not sort. Specify this value if you do not need sorting. In this case directory opening can be much faster especially for large directories.

The final parameter `dir_number` should be an expression evaluating to an integer value between 1 and 512. This number should be used as argument to the function `NextFile` and as parameter to the commands `CLOSE DIRECTORY` and `RESET DIRECTORY`. If this parameter is a variable having an integer value zero then the command will alter the value of the variable assigning a directory number automatically. This is similar to the mechanism when opening files. There is **no** `FreeDir` or any similar function.

If the directory does not exist or is not readable by the program for security or for any other reason an error occurs.

Note that the number used to open a directory is separate from the numbers used in opening files. If you use for example the file number 14 to open a file you still can use 14 to open a directory.

File and directory names are collected into a list during the command `OPEN DIRECTORY`. Any alteration in the file system like deletion of files, new files, modification of files will be reflected in the listing only if the directory is closed using the statement `CLOSE DIRECTORY` and reopened again using the statement `OPEN DIRECTORY`.

12.14.2 Function NextFile

When a directory is successfully opened the function `NextFile` can be used to retrieve the file names from the list. The argument to this function is an integer number, the directory number used to open the directory. The function returns a string value. When the last file name was retrieved the function returns `undef`.

12.14.3 Function EOD

The function `EOD` serves for End Of Directory checking. The argument to this function is the directory number used in the command `OPEN DIRECTORY`. The result of the function is `TRUE` if there are no more files to be returned by the function `NextFile` and `FALSE` otherwise.

12.14.4 Reset directory

You can use this command to reset the directory listing. The syntax of the command is

```
RESET DIRECTORY dn
```

or

```
RESET DIRECTORY #dn
```

where `dn` is the directory number. After executing this command the next call to `NextFile` will return the first file in the list and listing starts again.

12.14.5 Close directory

You can use this command to close the directory listing. The syntax of the command is

```
CLOSE DIRECTORY dn
```

or

```
CLOSE DIRECTORY #dn
```

Using this command you can free the space and resource allocated to the directory listing. Directory listing may need huge amount of memory in case of recursive listing of large directory structures. After using this command the directory number `dn` can be used again to open another or the same directory.

13 Networking

13.1 Opening a Socket

Networking in ScriptBasic is very simple. As you can open a file on the local disk you can open a service on a remote machine and print into it and get characters from it.

When you want to open a file to a service on a remote machine you should specify the name or the IP number of the remote machine and the port number in the open statement as a file name and you should open the file with mode socket. For example:

```
OPEN "www.digital.com:80" FOR socket AS 1
```

will open a connection to the web server of the machine named `wwww.digital.com` . The port number is `80` in this example, which is the usual port of a web server. You can also write:

```
open "16.193.50.33:80" for socket as 1
```

specifying the IP number of the machine. (The IP number listed in this documentation is an example and may not be the IP number of any server.) After opening a socket you can print into the channel and read from it using the commands `print`, `printnl`, `line input` and the function `input`. To retrieve a simple web page the following sample program can be used:

```
on error goto ErrorLabel
 open "16.193.50.33:80" for socket as 1
 print#1,"GET http://localhost/ HTTP/1.0\n\n"

 while not eof(1)
  line input#1,a
  print a
 wend
close 1
 stop
ErrorLabel:
 print "The web server 16.193.50.33 on port 80 is not reachable\n"
```

The open statement generates an error in case the remote machine is not reachable or if the domain name can not be resolved. If the IP number is specified instead of the name of the remote machine ScriptBasic does not query the DNS system, but tries to connect to the remote machine immediately. This way you can connect to remote machines a bit faster but you risk that the machine IP number is changed. A machine is usually identified by its name. The IP number is a low-level identification that reflects the topological location of the machine on the Internet. If the host is moved from one service provider to another or some other technology changes or developments make it necessary the IP number of the machine is going to change. Therefore it is recommended to use the name of the machine if possible.

13.2 Getting the host name

ScriptBasic can easily get the name of the host the code is running calling the function `hostname`. This function returns the internet name of the host. For example the code fragment

```
print hostname,"\n"
```

prints `vphome` on my machine.

14 String handling commands

14.1 Split and splita

The commands `split` and `splita` split a string into sub-strings. The syntax of the commands are

```
SPLIT expression BY expression TO variable_list
SPLITA expression BY expression TO variable
```

The first `expression` on the command is the string to split up. The second expression is the string used to split up the first one. For example

```
const nl="\n"
split "first,second,third,fourth" by "," to fi,se,th,fo
print fi,nl,se,nl,th,nl,fo
```

splits the string by the commas and will print

```
first
second
third
fourth
```

The resulting sub-strings get into the variables listed after the keyword to. In case of command `SPLITA` there is only a single variable after the keyword `TO`. This variable will become an array and the elements of the array will hold the sub-strings. The resulting array will be indexed starting with zero. Thus

```
splita "first,second,third,fourth" by "," to q
print lbound(q)," ",ubound(q)
```

will print

```
0 3
```

and the sub-strings get into the array `q[0]`, `q[1]`, `q[2]` and `q[3]`. In this case the array will have as many elements as need by the split string. If the string to be split is `undef` the result will also be `undef`. In this case the variable will not become an array, but remain simple variable holding the value `undef`.

If there are more variables in a split statement than sub-string the rest of the variables will become `undef`.

```
split "q,w,e" by "," to q,w,e,r,t
print r," ",t
```

will print

```
undef undef
```

If there are less number of variables than sub-strings the last variable will hold the remaining part of the original string holding the separator strings as well. For example:

```
split "q,w,e" by "," to q,w
print q,"\n",w,"\n"
```

will print

```
q
w,e
```

This way you can write programs that chop off few elements from a string containing some kind of list and leave the rest of the string in a variable. For example:

```
ListVar = "1,2,3,4,5,6,7,8,9,10,55"
while IsDefined(ListVar)
split ListVar by "," to FirstMember,ListVar
print FirstMember,"\n"
wend
```

will print

```
1
2
3
4
5
6
7
8
9
10
55
```

If two separator strings follow each other then the resulting sub-string is empty string. For example:

```
splita "1,,2" by "," to q
for i=lbound(q) to ubound(q)
print q[i],"\n"
next i
```

will print

```
1

2
```

However separator strings on the start and on the end of the string are ignored:

```
splita ",1,2,3," by "," to q
for i=lbound(q) to ubound(q)
print q[i],"\n"
next i
```

will print

```
1
2
3
```

The separator string can be any length, arbitrary string.

After a string is split into sub-string there is a need many times to put the parts together. To do that the string function `JOIN` can be used.

14.2 Unpack

The command unpack should be used to split a binary record into variables. A binary record is a string that contains integer and real numbers and strings inside as sub-strings. For more information see the detailed documentation of `UNPACK` in the ⟨undefined⟩ [Command reference], page ⟨undefined⟩.

15 Conditional Execution

Conditional execution is a vital point in any programming language. Conditional execution can be performed using the statement `IF`. This statement has the following format:

```
If expression then
 ' Code to execute when the
 ' expression is true
End if
```

Or it may have the format:

```
If expression then
 ' Code to execute when the
 ' expression is TRUE
Else
 ' Code to execute when the
 ' expression is FALSE
End if
```

Or even it may have the format:

```
If expression then
 ' Code to execute when the
 ' expression is TRUE

Elseif expression2 then
 ' Code to execute when the
 ' expression2 is TRUE
End if
```

And even the format:

```
If expression then
 ' Code to execute when the
 ' expression is TRUE
Elseif expression2 then
 ' Code to execute when the
 ' expression2 is TRUE
Else
 ' Code to execute when both
 ' expression and expression2 are
 ' FALSE.
End if
```

When closing a conditional execution construct with the keyword `ENDIF` you can write it as one word or as two separate words, like `END IF`. Also you have a great freedom spelling the keyword `ELSEIF`. You can use any of the followings: `ELSE IF` (two words), `ELSEIF`, `ELSIF` or `ELIF`.

Most BASIC languages have the single line format of the `IF` statement where the conditionally executed command is on the same line as the command `IF` and there is no `ENDIF` required. You can have the same in ScriptBasic. You can write:

```
IF expression THEN command
```

In this case the command is executed only if the expression standing after the keyword IF is `TRUE`. Unfortunately you can not have ELSE in this case. If you need the `ELSE` branch then use the multi line format of the conditional execution closing the `ELSE` branch with `ENDIF`.

When using the single-line format of the command `IF` there are some restrictions on the command that follows. The command executed conditionally can not be any declaration type command, like `Const`, `Module` or `Declare Sub` and can not be starting or ending a loop. Finally this command can not be another `IF, ELSEIF, ELSE` or `ENDIF`.

16 The statement GOTO

The statement `GOTO` is the most famous statement of all BASIC languages. Many program theorists say that you should not ever use `GOTO` and they may be right. Even so, the statement `GOTO` is part of most programming languages and ScriptBasic is no exception.

Using the statement `GOTO` you can alter the execution order of the statements. `GOTO` statements use labels and the labels can identify program lines. The form of a `GOTO` statement is

```
GOTO label
```

where the `label` is a label, which is defined somewhere in a program.

Labels should stand on the start of a line preceding a command. There are two different label types in ScriptBasic. One is the conventional BASIC style label, a decimal number. Any line can contain an unsigned integer as a label at the start of the line. For example the following code is perfectly legal in ScriptBasic:

```
10 print "Hello word!!"
20 REM
30 print "This is line 30"
```

These types of line labels are available in ScriptBasic for compatibility reasons, and there is no need to label each line in ScriptBasic. The numeric labels can be present in any order and need not increase. This is not compatible with old BASIC languages, but there is little need for that. In those days these numbers were needed to help the simple built-in editor to sort lines and to allow the programmer to insert in new lines. (Did you program those days when the ZX80, ZX Sinclair Spectrum, Atari 800XL, Commodore, Enterprise computers were on top?) If you use these numeric labels the lines will be executed in the order as they appear in the source file and not necessarily in increasing label order.

The type of label, which is more modern and lets the programmer to write more readable and as such easy to maintain programs are the alphanumeric labels. These labels follow the syntax of any identifier of ScriptBasic, stand on the start of the line and are followed by a colon. Both numeric labels and alphanumeric labels can be used in `GOTO` statements.

```
10 print "hah"
line input a
if a = "n\n" then
  goto finish
end if
goto 10

finish:
 REM this is the end of the program
```

This program loops printing three characters and waiting for input after each print so long as long the user types a single n character. When the user types the n character the program executes the statement `GOTO` to jump onto the label finish.

The labels are subject to name space modifications as any other variable or user defined function names as described in the section See ⟨undefined⟩ [Name spaces], page ⟨undefined⟩.

All `GOTO` instructions should reference a local label. This means that you can not jump out or jump into a function or subroutine code. You are free to reuse label names once in each function or subroutine. In other words if you use a label in a function you can use the same name as a label without worry in another function or subroutine as well as you can use it in global program code outside of all functions and subroutines once. All these labels are different ones although all may have the same name.

All the labels are local to the subroutine or function that the label is defined in. This is implemented in ScriptBasic via name decoration. If you declare a label named `MyLabel` on a line ScriptBasic automatically converts it to `module::MyLabel'function`, where module is the name of the actual module (this is `main` if no module is defined); `function` is the name of the actual function or subroutine without the current module name. If the label is in the global program code outside of all functions and subroutines the function name is empty string. In this case the label has the decorated form `module::MyLabel'`. The modules can reference each-others labels using explicit module name notation. However there is no possibility to reference any label, which is not in the same function or subroutine.

17 Loop constructs

ScriptBasic has a rich set of looping constructs. If you got used to a special construct you can have it in ScriptBasic. The looping constructs that ScriptBasic supports are:

```
While expression
 REM loop body
Wend
```

Repeat the loop body so long as long the expression is `TRUE`. Do the test before entering the loop.

```
Repeat
 REM loop body
Until expression
```

Repeat the loop body so long as long the expression is `FALSE`. The loop body is executed at least once and the test is performed after the execution of the loop body.

```
Do While expression
 REM loop body
Loop
```

This is just another form of the loop while/wend. Repeat the loop body so long as long the expression is `TRUE` and do the testing before the loop body execution.

```
Do Until expression
 REM loop body
Loop
```

Repeat the loop body so long as long the expression is `FALSE`. Do the testing before the loop body execution.

```
Do
 REM loop body
Loop While expression
```

Repeat the loop body so long as long the expression is `TRUE`. Do the testing after the loop body execution.

```
Do
 REM loop body
Loop Until expression
```

Repeat the loop body so long as long the expression is `FALSE`. Do the testing after the loop body execution.

```
FOR variable=StartValue TO EndValue STEP StepValue
 REM loop body
NEXT
```

This loop loads the value of the `StartValue` into the *variable*. After the execution of the loop the variable is increased by the value of `StepValue`. The `StepValue` can also be negative, in which case the value of the variable will decrease. The loop is repeated so long

as long the value reaches or steps over the `EndValue`. If the `StartValue` is already over the `EndValue` the loop is never executed.

The keyword `STEP` and the value after the keyword is optional. In this case the step value is one.

18 Functions and Subroutines

Most programming languages provide a form for defining code fragments to be used later at several times. These are subroutines and functions. ScriptBasic allows the programmer to define subroutines before or after they are used; to have local variables inside functions and subroutines and allows the functions and subroutines call each other even recursively.

18.1 Declaration of subroutines and function

You can declare a function like

```
FUNCTION MyFunction(var1, var2, var3)
 REM function body
END FUNCTION
```

You can call the function in any expression, like

```
A = MyFunction(1,2,3)
```

Subroutine declaration is almost the same:

```
SUB MySub(Var1,Var2, Var3)
 REM subroutine body
END SUB
```

You can call the subroutine using the CALL statement:

```
CALL MySub(1,2,3)
```

The major difference between functions and subroutines is that subroutines do not have values returned. However ScriptBasic does not differentiate between functions and subroutines. `SUB` and `FUNCTION` are just two keywords that you can use almost interchangeable. A subroutine can return a value and functions may be called using the call statement, although this is not recommended.

18.2 Calling functions and subroutines

Functions and subroutines can be called two different ways. One way is to use the function in an expressions the same way as you would do with the built-in function like `sin`, `cos` or `rnd`. Calling a function this way you have to use the name of the function and supply parameters between parentheses.

Another way to call a function or subroutine is to use the call statement. The call statement has a very simple syntax:

```
CALL function_name( argument_list )
```

The function name is the name that stands after the keyword function or sub. The argument list is a comma-separated list of expressions. When you call a function in a call statement you can omit the parentheses, which is very convenient when you do not have arguments. Therefore the following lines are equivalent:

```
CALL MyFunction
CALL MyFunction()
```

or

```
CALL MyFunction 1,2,3
CALL MyFunction(1,2,3)
```

To ease readability and programmers life you can even omit the keyword `CALL` if the function or subroutine was already defined. Therefore you can write:

```
Sub MyFunction
Print "I am in my function\n"
End sub

MyFunction
```

On the other hand the following code

```
MyFunction

Sub MyFunction
Print "I am in my function\n"
End sub
```

is not valid, because the function is not defined when it is called. In such a situation you have to use the keyword `CALL`.

If the function or subroutine has formal arguments you can pass values inside the parentheses. Functions and subroutines can have as many arguments as you like and you can pass as many values as you like. ScriptBasic does not check that the number of actual values passed as argument is the same as the number formal arguments. If you pass more arguments than the number of the formal arguments, the last values are calculated and the result is thrown away. Have a look at the following example:

```
a = MyFunction(1,2,3)
a = MyFunction(1,2)
a = MyFunction(1,2,3,MyFunction("haha", "hehe","hihi"))
print a
printnl
function MyFunction(a,b,c)
local x

print a,b,c,x


printnl


end function
```

And the output printed to the screen:

```
123undef
12undefundef
hahahehehihiundef
123undef
undef
```

The lines 1 to 4 are results of a function call. When we call the function the first time we pass three values to the arguments, which are named *a*, *b* and *c*. The values are printed correct and the value of the local variable *x* is `undef`.

The second time we call the function passing only two arguments. ScriptBasic does not complain. As we do not pass any value for the argument named *c* the value of it is `undef`.

When we call the function third time the argument evaluation calls the function itself again. Although this is the fourth argument and `MyFunction` has only four this argument is evaluated and this result the line 3 in the output. The result of this function call, which is `undef` anyway, is not used.

18.3 Returning a value

The example above did not return any value. But functions are to return values. To return a value from a function the code should perform an assignment that looks very similar to a normal assignment statement. The difference is that the "variable" that stands on the left side of the = should be the name of the function. Let's see the previous example a bit extended:

```
a = MyFunction(1,2,MyFunction(1,1,1))
print a
printnl
function MyFunction(a,b,c)
local x

print a,b,c,x
printnl
x = a * b * c
MyFunction = a + b + c

end function
```

The output is

```
111undef
123undef
6
```

The value of the function call is 3 when the function is first executed. This value is passed to the function call itself as third argument; and is used to calculate the final values for the variable a.

18.4 Local and global variables

In the example the local variable *x* is assigned a value, but is never used. Local variables in ScriptBasic serve the same purpose as in other languages. They are to store values for the time of function or subroutine execution. They are automatically created when a function or subroutine starts and vanish as the subroutine or function finishes its task.

Local variables are the only variables that have to be defined in ScriptBasic. If a variable is not declared as local then ScriptBasic will think that the variable is global. Global variables keep their values while the program executes. See the following example:

```
call MySub()
call MySub()
sub MySub
local x

print x," ",y,"\n"
x = 3
y = 4
end sub
```

The output of the program is:

```
undef undef
undef 4
```

The variable y is not defined when the subroutine is called the first time. The variable x is local and is also not defined. When the program calls the subroutine the second time the local variable x is freshly created and therefore has no defined value. On the other hand the global variable has the value assigned to it during the previous subroutine call.

Local variables not only exist to create new local space. They can also help the programmer to protect global variables. A function or subroutine may use a variable named ExampleVariable and may alter the value of it. Other parts of the program may use the same variable without knowing that the variable value changes in the function or subroutine call. For example the program:

```
ExampleVariable = 3
call MySub()
print ExampleVariable
sub MySub
ExampleVariable = 5
end sub
```

prints out the value 5. If we alter the program

```
ExampleVariable = 3
call MySub()

print ExampleVariable
sub MySub
local ExampleVariable
ExampleVariable = 5

end sub
```

the output becomes 3. The variable *ExampleVariable* is local in this second case and it does not interfere with the global variable of the same name.

18.5 More on local and global variables

Although ScriptBasic following the good old BASIC conventions implicitly defining all variables not declared as local to be global inside subroutines and function this is not always a good approach. This is quite convenient in case of small programs, but may harm when writing large programs. One programmer may sloppily write a function that uses the variable `I` as a loop variable in a `FOR` loop. This is quite common. Not declaring this variable to be `local` is also common. Debugging days finding out why the global variable `I` is changing in a code fragment calling a function which is calling a function which is calling the function containing the loop is also quite common.

As you could see there is a `declare option` command that requires the programmer to declare all variables, but it does not solve this problem.

The solution in more modern languages is that all variables inside functions and subroutines are local unless they are explicitly declared to be global. ScriptBasic can also follow this way. To do this you have to insert the command:

```
declare option DefaultLocal
```

This will mean that all variables inside a function or subroutine following this line will be local unless explicitly declared to be global. This can be used to prevent programming bugs like the one described above. This will make the

18.6 Parameters passed by value and by reference

When you call a function or subroutine and pass variables as arguments do you pass the value of the variable or the variable itself. Look at the following simple example:

```
a = 1
call MySub(a)
print a
sub MySub(x)
x = x + 1
end sub
```

Will it print 1 or 2? The answer is that it will print 2. **ScriptBasic passes the variables and not the value of the variable whenever possible.** If we alter the program a bit and write

```
a = 1
call MySub(ByVal a)
print a
sub MySub(x)
x = x + 1
end sub
```

the resulting output will be 1. Here the difference is the use of the operator `ByVal`. This operator is an identity operator that does nothing, but this doing nothing is valuable. The result of this "nothing" is that the passed value becomes an expression and is not a variable anymore. As such can not be passed by reference only by value the function can not alter the value of the variable *a*.

18.7 ByVal command

As you could see the caller can alter the behavior of argument passing forcing the argument to be passed by value instead of reference. This can also be done inside the called function or subroutine. To do this the pair of the `ByVal` operator, the `ByVal` command, can be used. The syntax of the command is very simple:

```
ByVal variable list
```

The variables listed in the command become all passed by value instead of passed by reference and can be altered without modifying the caller variable. For example

```
a = 1
call MySub(a)
print a
sub MySub(x)
ByVal x
x = x + 1
end sub
```

Will print 1, because the local variable *x* gets the value of *a* and not the reference to *a*. Note however that this is a command executed by ScriptBasic and is not a declaration. Therefore

```
a = 1
call MySub(a)
print a
sub MySub(x)
x = x + 1
ByVal x
X = x + 1
end sub
```

will print 2. Why? Because the first increment is done on the reference to *a*, and therefore *a* is incremented. After this increment the command `ByVal` is executed. This command evaluates the variables listed after it and assigns the values to the variables. The only effect is that the variables now contain the value copied from the originally referenced variable and not a reference to the caller variable.

18.8 Calling functions indirectly

Whenever you call a function or subroutine you have to know the name of the subroutine or function. In some situation programmers want to call a function without knowing the name of the function. For example you want to write a sorting subroutine that sorts general elements and the caller should provide a subroutine that makes the comparison. This way the sorting algorithm can be implemented only once and need not be rewritten each time a new type of data is to be sorted.

The sorting subroutine gets the comparing function as an argument and calls the function indirectly. ScriptBasic can not pass functions as arguments to other functions, but it can

pass integer numbers. The function ADDRESS can be used to convert a function into integer. The result of the built-in function ADDRESS is an integer number, which is associated inside the basic code with the function. You can pass this value to the ICALL command or function as first argument. The ICALL command is the command for indirect subroutine call. The call

```
ICALL ADDRESS(MySubroutine()),arg1,arg2,arg3
```

is equivalent to

```
CALL MySubroutine( arg1,arg2,arg3)
```

If you call a function that has return value use can use the ICALL function instead of the ICALL statement:

```
A = ICALL(ADDRESS(MyFunction()),arg1,arg2,arg3)
```

is equivalent to

```
A = MyFunction(arg1,arg2,arg3)
```

The real usage of the function ADDRESS and ICALL can be seen in the following example:

```
sub MySort(sfun,q)
local ThereWasNoChange,SwapVar
repeat
 ThereWasNoChange = 1
 for i=lbound(q) to ubound(q)-1

  if  icall(sfun,q[i],q[i+1]) > 0 then
   ThereWasNoChange = 0
   SwapVar = q[i]
   q[i] = q[i+1]
   q[i+1] = SwapVar
  endif

 next i
until ThereWasNoChange

end sub


function IntegerCompare(a,b)
  if a < b then
   cmp = -1
  elseif a = b then
   cmp = 0
  else
   cmp = 1
  endif
end function

h[0] = 2
h[1] = 7
```

```
h[2] = 1

MySort address(IntegerCompare()) , h

for i=lbound(h) to ubound(h)
 print h[i],"\n"
next i
```

Note that the argument of the function ADDRESS is a function call and not the name of the function. In other words the argument of the function ADDRESS is the name of the function and the opening and closing parentheses. ScriptBasic allows variables and functions to share the same name. ADDRESS is a built-in function just as any other built in function, and therefore the expression

```
Address(MySub) THIS IS WRONG!
```

is syntactically correct. The only problem is that it tries to calculate the address of the variable MySub, which it can not and results a run-time error. Instead you have to write

```
Address( MySub() )
```

using the parentheses. In this very special situation the function or subroutine MySub() will not be invoked, because the built-in function ADDRESS does not start it. The parentheses needed only to tell the compiler that this is a function and not a variable.

18.9 GOSUB and RETURN

Most basic language implementations contain the commands GOSUB and RETURN. ScriptBasic is no different. This command pair is the old form of subroutine calling that existed in the basic language before functions and subroutines were invented. When a GOSUB command is executed there is no new function or subroutine started with local variables and return value. This is almost the same as executing the command GOTO. You have to specify where the execution of the code is to be continued using a label and the execution will go on at that label. The only difference between GOTO and GOSUB is that GOTO forgets where it jumped to the specified label from. GOSUB on the other hand remembers and whenever a RETURN command is executed it jumps back to the command following the instruction GOSUB.

A code fragment started by a GOSUB command can use another GOSUB command any level deep and the command RETURN will always return to the command line following the last executed, matching GOSUB. For example:

```
print 1
gosub sub1
print 5
stop
sub1:
print 2
gosub sub2
print 4
```

```
return
sub2:
print 3
return
```

will print

```
12345
```

When a `GOSUB` command is executed inside a function or a "real" subroutine that starts with the command `SUB` the label following the keyword `GOSUB` can not be outside the actual function or subroutine. It is not a must to return from a code segment started with `GOSUB`. If there is a `GOSUB` in the program and the program finishes before reaching the pairing `RETURN` the program just finishes normally. This is not an error. Also when a `function` or `sub` finishes before executing a `RETURN` command for an already executed `GOSUB` in that `function` or `sub` the execution simply returns to the following command where the function or sub was called and the unpaired `GOSUB` return address or addresses are dropped. For example:

```
sub sub1
print 3
gosub kukk
print "never printed"
exit sub
kukk:
print 4
end sub

print 1

gosub obenal
print 6
stop
obenal:
print 2
call sub1
print 5
return
```

will print

```
123456
```

19 Reference Variables

Even if you did not realize, you have already met reference variables in ScriptBasic. These are variables, which have no values for themselves but rather reference another variable and whenever they are accessed the other variable is accessed actually (With some few exceptions, which are cases that help handling reference variables.)

The arguments of functions and subroutines are examples. These variables refer to the variable that stands in the calling expression. For example:

```
sub myfunction(a)
  a = a + 1
end sub
b = 3
myfunction b
```

will increase the value of the global variable *b*, because *a* references this variable.

This is pretty automatic and there would be no need to devote a special chapter to such a phenomenon, but there is more. Not only argument variables of functions and subroutines can be reference variables but any variable using the reference assignment. This is almost like a normal assignment. The difference in the syntax is that the command starts with the keyword `REF` and the right side of the assignment can not be just any expression, but a variable that the left side is going to reference. For example:

```
a = 13
REF b = a
b += 1
print a
```

will print `14` because *b* is the "same" as *a* after the `REF` assignment.

TO BE CONTINUED... ***TODO***

20 Pattern matching

ScriptBasic provides a simple, yet powerful pattern matching mechanism that eases string handling. ScriptBasic pattern matching is based on wild card patterns and not regular expressions. There is an external module called `RE` that implements regular expression based pattern matching.

The pattern matching that ScriptBasic provides is simpler and therefore easier to learn and use. Most programs do not need the power of regular expression match.

20.1 The operator LIKE

Pattern matching in ScriptBasic is similar to the pattern matching that you get used to on the UNIX or Windows NT command line. The operator `LIKE` compares a string to a pattern.

```
string LIKE pattern
```

Both `string` and `pattern` are expressions that should evaluate to a string. If the pattern matches the string the result of the operator is `TRUE`, otherwise the result is `FALSE`.

The pattern may contain normal characters, wild card characters and joker characters. The normal characters match themselves. The wild card characters match one or more characters from the set they are for. The joker characters match one character from the set they stand for. For example:

```
"file.txt" like "*.txt" is TRUE
"file0.txt" like "*?.txt" is TRUE
"file.text" like "*.txt" is FALSE
```

The wild card character `*` matches a list of characters of any code. The joker character `?` matches a single character of any code. In the first print statement the `*` character matches the sub-string `file` and `.txt` matches `.txt` at the end of the string. In the second example `*` matches the string `file` and the joker `?` matches the character `0`. The wild card character `*` is the most general wild card character because it matches one or more of any character. There are other wild card characters. The character `#` matches one or more digits, $ matches one or more alphanumeric characters and finally `@` matches one or more alpha characters (letters).

`*` all characters

`#` 0123456789

`$` 0123456789abcdefghijklmnopqrstxyvwzABCDEFGHIJKLMNOPQRSTXYVWZ

`@` abcdefghijklmnopqrstxyvwzABCDEFGHIJKLMNOPQRSTXYVWZ

A space in the pattern matches one or more white spaces, but the space is not a regular wild card character, because it behaves a bit different.

Note that wild card character match ONE or more characters and not zero or more as in other systems. Joker characters match exactly one character, and there is only one joker character by default: the character ?, which matches a single character of any code.

20.2 The function JOKER

We can match a string to a pattern, but that is little use, unless we can tell what sub-string the joker or wildcard characters matched. For the purpose the function JOKER is available. The argument of this function is an integer number, n starting from 1 and the result is the sub-string that the last pattern matching operator found to match the n-th joker or wild card character. For example

```
Const nl="\n"
if "file.txt" like "*.*" then
  print "File=",joker(1)," extension=",joker(2),nl

else

  print "did not match"
endif
```

will print

```
File=file extension=txt
```

If the pattern did not match the string or the argument of the function `JOKER` is zero or negative, or is larger than the serial number of the last joker or wild card character the result is `undef`.

Note that there is no separate function for the wild card character sub-strings and one for the joker characters. The function `JOKER` serves all of them counting each from left to right. The function `JOKER` does not count, nor return the spaces, because programs usually are not interested in the number of the spaces that separate the lexical elements matched by the pattern.

20.3 Escaping wild cards

Sometimes you want a wild card character or joker character to match only itself. For example you want to match the string `"13*52"` to the pattern two numbers separated by a star. The problem is that the star character is a wild card character and therefore `"#*#"` matches any string that starts and ends with a digit. But even if that is not a problem more issues still may arise. A `*` character matches one or more characters, and therefore `"#*#"` will indeed match `"13*52"`. The problem is, when we want to use the sub-strings.

```
Const nl="\n"
a="13*52" like "#*#"
print joker(1)," ",joker(3),nl
a="13*52" like "#~*#"
print joker(1)," ",joker(2),nl
```

will print

```
1 52
13 52
```

The first `#` character matches one character, the `*` character matches the sub-string `"3*"` and the final `#` matches the number `52`. (We will soon explain the details why that is in the section #refAmbiguous matching.)

The solution is the pattern escape character. The pattern escape character is the tilde character: `~`.

Any character following the `~` character is treated as normal character and is matched only by itself. This is `TRUE` for any normal character, for wild card characters; joker characters; for the space and finally for the tilde character itself. The space character following the tilde character matches exactly one space characters.

Later you will learn that `"#*#"` may also serve the purpose, when the role of the wild card character `*` is changed.

20.4 Ambiguous matching

Pattern matching is not always as simple as it seems to be from the previous examples. The pattern `"*.*"` matches files having extension and `joker(1)` and `joker(2)` can be used to retrieve the file name and the extension. What about the file `sciba_source.tar.gz`? Will it result

```
File=scriba_source.tar extension=gz
```

or

```
File=scriba_source extension=tar.gz
```

?

The correct result is the second. Wild card characters implemented in ScriptBasic are not greedy. They eat up only as many characters as they need. If you need more flexibility then you have to use the more complex module `RE` that implements regular expression search and replace.

20.5 Advanced matching

Up to now we were talking about wild card characters and the joker character defining what matches what as final rule carved into stone. But these rules are only the default behavior of these characters and the program can alter the set of characters that a joker or wild card character matches.

There are 13 characters that can play joker or wild card character role in pattern matching. These are:

```
*  #  $   ?  &  %  !  +  /  |  <  >
```

When the program starts only the first five characters have special meaning the others are normal characters. To change the role of a character the program has to execute a `SET JOKER` or `SET WILD` command. The syntax of the commands are:

```
SET JOKER expression TO expression
SET WILD expression TO expression
```

Both expressions should evaluate to string. The first character of the first string should be the joker or wild card character and the second string should contain all the characters that the joker or wild card character matches. The command `SET JOKER` alters the behavior of the character to be a joker character matching a single character in the compared string. The command `SET WILD` alters the behavior of the character to be a wild card character matching one or more characters in the compared string. For example if you may want the `&` character to match one or more of all hexadecimal characters the program has to execute:

```
SET WILD "&" TO "0123456789abcdefABCDEF"
```

If a character is currently a joker of wild card character you can alter it to be a normal character issuing one of the commands

```
SET NO JOKER expression
SET NO WILD expression
```

where expression should evaluate to a string and the first character of the string should give the character to alter the behavior of.

The two commands are identical, you may always use one or the other; you can use `SET NO JOKER` for a character being currently wild card character and vice versa. You can execute the command even if the character is currently a normal character in the pattern matching game.

Using the commands now we can see that

```
Const nl="\n"
Set no wild "*"
a="13*52" like "#*#"
print joker(1)," ",joker(2),nl
```

will print

```
13 52
```

giving the desired result.

If the expression supposed to result the character for which the role and character set is defined is none of the 13 characters listed above an error occurs.

21 Handling run-time errors

Tao says: Errors happen.

Old programming languages like C handle error conditions returning `NULL` pointer from a function, or returning zero, or returning non-zero and applying system specific library functions, like `longjump`, `setlongjump` and alike. Newer programming languages, like C++, Java invented exception handling.

BASIC is an old language. Indeed it is a very old language. Older than C. But it *has* exception handling.

When an error happens during program execution the program execution stops (exit code is the code of the error or zero on errorless termination) and usually an error message is printed on the screen. This happens for example when a file can not be opened for reading, there is not enough memory to perform some operation, an external module can not be loaded for some reason, a specified file number is out of range and many other events can cause an error.

A programmer can have two different approaches to avoid program termination caused by program error. One approach is to check every needed condition before trying to perform some action. This is impractical in some cases. The other approach is to tell the BASIC interpreter what to do instead of terminating the program when some error occurs. Programmers usually follow a mixed style. Some conditions are easier to check, while other errors are difficult to avoid.

For example you can not check all the conditions before trying to load a module. To do that you have to check the BASIC setup, configuration, the existence of the file, the existence of the functions implemented in the file. It is not possible from basic.

21.1 On error goto

The error handling instructions that ScriptBasic have are the usual and well known `ON ERROR GOTO` constructs. The easiest example using this construct is:

```
ON ERROR GOTO ErrorHappens

ERROR 1

PRINT "This won't print\n"
ErrorHappens:
PRINT "This is the error message.\n"

END
```

will print

```
This is the error message.
```

The error is caused by the statement error, which artificially generates an error of the code given on the line after the keyword. The statement `ON ERROR GOTO` declares where to continue execution when an error happens. The execution system remembers this and starts to execute the code after the label, when an error happens.

To switch off the effect of the `ON ERROR GOTO` statement you can execute a command

```
ON ERROR GOTO NULL
```

Note that other BASIC implementations use the label `0` and the form `on error goto 0`. However in ScriptBasic this means to jump to the label `0` in case of an error. This label is completely valid in ScriptBasic though its use is not recommended. The form

```
ON ERROR GOTO NULL
```

used in ScriptBasic is more readable and expresses the real meaning not to jump anywhere in case of error, but rather handle the error the normal way.

21.2 Resume

The piece of code executed when an error happens is usually tries to repair the condition that caused the error. When the reparation is done the program has to resume its normal operation. To perform it the program can use the statement `RESUME`. This statement has three forms.

```
RESUME
```

continues the execution at the line that caused the error. In other words the line is executed again.

```
RESUME NEXT
```

continues the execution of the program after the line that caused the error. This means that the program does not try to execute the line that caused the error. Finally the statement

```
RESUME label
```

resumes the program execution at the label specified after the keyword.

The interpreter remembers the resume point when an error occurs. After execution the statement `RESUME` this resume point is cleared and the last error code is set to zero. In other words the execution returns from error correction to normal operation.

Although the statement `RESUME` clears the last error value, there is another way to clear this value. You can execute the statement

```
ERROR 0
```

This causes an artificial error of code zero, which means no error. This also sets the last error code to zero meaning no error. Although the statement `ERROR 0` seems to nullify an error this does not switch execution to normal from error correction, because still there is a resume point remembered.

Executing a resume statement in normal operation, when there is no resume point remembered causes an error.

21.3 RESUME

If you know that the program error does not need error code you can use one of the `ON ERROR RESUME` statements. These have almost the same effect as the statement `ON ERROR GOTO` except that the program execution continues in non-error mode when an error happens. This means that the program jumps to the location specified by the statement `ON ERROR`

RESUME, but the error code is zero meaning no error and there is no remembered error location where a statement RESUME could return.

There are two different forms of the statement ON ERROR RESUME. One is similar to the statement ON ERROR GOTO specifying a label where to resume execution after the error. This has the form

```
ON ERROR RESUME label
```

The other type has the form

```
ON ERROR RESUME NEXT
```

This command tells the interpreter to neglect the erroneous line and continue the operation executing the next line. Although this is a pleasant and easy way handling error, great care has to be taken. If you use the statement ON ERROR RESUME NEXT in a code an error may silently be passed. On the other hand if there are more than one errors in the code the second one will terminate the program execution, because the first error switches off the effect of the statement ON ERROR RESUME NEXT.

The code executing the error correction code can access the error code. The code of the last error happened is returned by the function ERROR(). This function has no argument and returns an integer value, which is the error code. In normal operation, outside of error correction code the value of this function is zero.

21.4 Levels of error handling

The error correction code should be simple and carefully designed. If an error happens while executing the error correction code the interpreter will terminate the program code or at least will go up to the next level of exception handling.

When the interpreter executes a jump to the error correction code caused by an error the ON ERROR GOTO functionality is switched off automatically until a new ON ERROR GOTO instruction is executed. This is done to prevent infinite looping caused by errors in the error correction code. The error correction code therefore usually executes an ON ERROR GOTO statement pointing to the start of the error correction code itself immediately before the execution the statement RESUME.

At the same time an error correction code can not have its own error correction code. If an error correction code uses the ON ERROR GOTO statement to specify its own error correction code it looses the resume point when an error occurs. In other words the remembered point of code where to resume after error correction is always the statement where the last error occurred. There is only a single resume point during execution and there is no stack of resume points.

The error handling statements use labels. Labels in ScriptBasic are local. You can not refer to a label in a function or subroutine from outside and vice versa. There is no need to do so with the error handling functions as well. The issue is that the jumping instruction is executed some time after the label was used.

What happens when an statement ON ERROR GOTO is used in function X and the error happens in function Y called from function X. Will the code jump from one function to the other?

```
sub X
on error goto ErrorLabel
print "starting sub Y\n"
call Y
print "sub Y returned\n"
exit sub
ErrorLabel:
print "an error occurred while executing Y\n"
end sub

sub Y

error 1
end sub

call X
end
```

will print

```
starting sub Y
an error occurred while executing Y
```

The answer seems to be yes, but the case is not so simple. When the error occurs the interpreter sees that there is no `ON ERROR GOTO` statement currently in effect in the subroutine `Y`. Therefore it handles the error and terminates the execution of the subroutine `Y`. The subroutine returns and the error is inherited to the statement `CALL` invoking `Y`. The interpreter gets into the error condition again and now, at this level of execution there is an effective `ON ERROR GOTO` statement, which is executed.

To see that this really happens this way see the following example:

```
sub X
on error goto ErrorLabel
print "starting sub Y\n"
call Y
print "sub Y returned\n"
exit sub
ErrorLabel:
print "an error occurred while executing Y\n"
Flag = 0
Resume
end sub

sub Y
print "Y started\n"
if Flag = 1 Then
  error 1
end if
print "Y finishes\n"
end sub
```

```
Flag = 1

call X
end
```

will print

```
starting sub Y
Y started
an error occurred while executing Y
Y started
Y finishes
sub Y returned
```

In this code the subroutine `Y` causes error if the global variable *Flag* is 1. The error correction code sets this variable to zero and corrects the error with this action. After resuming the execution the code continues at the subroutine call and not at the error statement. This shows that the error condition does not jump out of the function. It rather terminates the function and propagates the error code up to the caller until the main program level is reached and program execution is terminated or there is an effective `ON ERROR GOTO` or `ON ERROR RESUME` statement.

When an `ON ERROR GOTO/RESUME` statement is executed it overrides the settings of the previously executed `ON ERROR GOTO/RESUME` statement, which is in effect. But it does not override the settings of any higher level settings, because those are not currently in effect and the settings are restored when the program returns from the actual function or subroutine. For example:

```
Global const nl ="\n"
sub ErrorSub
  on error goto ErrorLabel
  error 1
  print "No error has occurred in the function. Weird.\n"
  goto FinishLabel
ErrorLabel:
  print "An error has occurred inside the sub\n"
  print "and now generating another error, which is not

handled by the subroutine.\n"
  error 2
FinishLabel:

end sub

  on error goto ErrorLabel

  call ErrorSub

  print "No error"
  goto FinishLabel
```

```
  ErrorLabel:
  print "Error code: ",error(),nl

FinishLabel:
end
```

will print

```
An error has occurred inside the sub



and now generating another error, which is not


handled by the subroutine.

Error code: 2
```

Note that the `ON ERROR GOTO` setting of the module outside the subroutine remained valid although another `ON ERROR GOTO` setting was issued inside the subroutine.

21.5 Error codes

When an error happens the error code can be retrieved using the function `ERROR()` in the error handling code. This value is an integer value. To compare this value against a specific error code the program should include the file '`error.bas`'. This file defines global constants for each run time error. It is advised to use these symbolic constants because there is no guarantee that the actual value of an error code remains the same between builds. The file '`error.bas`' is generated automatically when the interpreter C language constants are generated. The file also contains the English language explanation of each error following the line that defines the global constant for the error code.

22 Setting options

ScriptBasic implements many features, operators, functions, and statements. The developers defined the behavior of a statement, function or operator. For example ScriptBasic has an equality test operator like any other language. This is the well know `=` sign that result `TRUE` if the two values compared are equal and `FALSE` otherwise. This is simple when the values are integer or real values. But what about strings? Should strings be compared case sensitive or case insensitive? Should the developers of ScriptBasic decide instead of you? Not at all.

The statement option allows the programmer to alter the behavior of certain features. For example you can say

```
OPTION COMPARE sbCaseInsensitive
```

and string comparison as well as pattern matching becomes case insensitive when the line is executed. To switch it to case sensitive the programmer can write

```
OPTION COMPARE sbCaseSensitive
```

and comparison is case sensitive again. Note that the option statement alters the behavior of the feature globally. No matter where the option statement is executed. It can be inside a function, somewhere in a module: it alters the feature for the program execution at top level.

The syntax of the instruction is:

```
OPTION symbol expression
```

`symbol` is any symbol that an instruction is looking at. Expression should result an integer value. Options are always integer values.

Also note that there is no checking if you misspell the symbol name. If you write

```
OPTION COMPARA sbCaseInsensitive
```

the statement will execute without error. It will set the option for the symbol `COMPARA` instead of `COMPARE` and the statement does not ever know that there is no one interested in the value associated with `COMPARA`. Option values are available for commands as well as for external modules.

You can set the value of the option `RaiseMathError`. This will alter the behavior of certain mathematical operators and mathematical functions. When an operator gets an operand or a functions and argument that is out of the scope of the operation or function it returns `undef` by default. In many cases this is inconvenient and error prone, because this way errors may propagate further. Using this option these operators alter their behavior. There are three constants that the programmer can use to alter argument handling:

`sbMathErrDiv` will result error when dividing with zero (division, integer division and modulo calculation.

`sbMathErrUndef` will result error when a mathematical operator gets `undef` as argument.

`sbMathErrUndefCompare` will result error when a comparison is done against `undef`.

These values can be joined together using the bit operator OR, for example

```
OPTION RaiseMathError sbMathErrDiv OR sbMathErrUndef
```

means that ScriptBasic will raise error when division with zero occurs or a mathematical operator gets `undef` as argument.

23 Other miscellaneous commands

23.1 Sleep

The sleep command causes the basic program to stop and wait for a few seconds. The number of seconds is given as argument. The syntax of the command is very simple:

```
SLEEP expression
```

The `expression` specifies the number of seconds to sleep. This is the only effective way to insert delay into the code. Old style waiting constructs, like empty for loops consume processor and are not precise. The command sleep calls the system function `sleep` (`Sleep` on Windows NT) and this allows other processes to get processor time while the basic program is sleeping.

Because nor UNIX operating systems neither Windows NT or Windows95/98 are real time systems there is no guarantee that the basic program is going to run immediately after the specified number of seconds has elapsed.

Because of system limitations the time to sleep should not be more than 2,147,483 seconds. This is approximately 24 days. If you need more you can create a loop around the sleep statement and call it many times. This wastes a bit of CPU but not too much. If the argument to the command sleep exceeds this limit the result is unknown.

To test the functionality of the command `SLEEP` run the following program:

```
for i=1 to 20
print i
print
sleep 1
next i
```

24 Using External modules

External modules are ScriptBasic extensions that are written in C and compiled to dynamic load libraries. ScriptBasic is capable loading these libraries and the functions implemented in the module become callable from Basic.

The line

```
DECLARE SUB basfun ALIAS "cfun" LIB "dlllib"
```

declares that the function `basfun` is implemented in an external library named `dlllib`. The name of the function that implements the library is called `cfun`. In your basic program you can call this function the same way as any other user-defined function or subroutine:

```
call basfun(arg1,arg2,...,argn)
```

or

```
a$ = basfun(arg1,arg2,...,argn)
```

The function name `cfun` is the name of the function as the dll or so object file exports it. The last string gives the name of object file. In the example above it is `dlllib`.

The dll object file name can be specified as an absolute file name or as a relative name. If the string specifies an absolute file name ScriptBasic tries to load the specified dll file. In this case the basic program has to specify the full path, the name of the file and the extension. If the name of the library file is a relative name it should not contain the path, or the extension. In this case ScriptBasic appends the default file extension for dynamic load libraries and tries to locate the files in the configured module libraries. The default file extension for dynamic load libraries and the module library directories are defined in the configuration file.

In case ScriptBasic can not fid the module in the module directories specified in the configuration file it tries under Windows to find the module dll in the same directory where the executable `scriba.exe` is and finally in the directory `..\modules` relative to the directory of the executable scriba.exe. For example if the installation directory for ScriptBasic is

```
C:\ScriptBasic
```

the default location of the executable is

```
C:\ScriptBasic\bin\scriba.exe
```

In this case

```
C:\ScriptBasic\bin\dlllib.dll and
C:\ScriptBasic\modules\dlllib.dll
```

are also tried. This allows you to start ScriptBasic programs without tedious installation procedures, editing the registry or setting up complex configuration file.

ScriptBasic is NOT capable calling arbitrary external functions implemented in a dll file. The functions should be implemented according to the calling conventions and parameter passing methods that ScriptBasic supports. External modules usually come with the appropriate basic include file that you should include into your basic program. The included files usually contain all the external function declarations and your job is to use the functions.

25 Command reference

25.1 ABS

Returns the absolute value of the argument. If the argument is a string then it first converts it to integer or real value. The return value is integer or real value depending on the argument.

`ABS(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.2 ACOS

Calculates the arcus cosine of the argument, which is the inverse of the function See ⟨undefined⟩ [COS], page ⟨undefined⟩. If the argument is not between (-1.0,+1.0) the return value is `undef`.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`ACOS(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.3 ACOSECANT

This is a planned function to calculate the arcus cosecant of the argument.

25.4 ACTAN

This is a planned function to calculate the arcus cotangent of the argument.

25.5 ADDDAY

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the day by the second argument and returns the time value for the same hour and minute but some days later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 86400 times the second argument to the first argument and returns the result.

25.6 ADDHOUR

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the hours by the second argument and returns the time value for the same minute and seconds but some hours later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 3600 times the second argument to the first argument and returns the result.

25.7 ADDMINUTE

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the minutes by the second argument and returns the time value for the same seconds but some minutes later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 60 times the second argument to the first argument and returns the result.

25.8 ADDMONTH

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the month by the second argument and returns the time value for the same day, hour and minute but some months later or sooner in case the second argument is negative.

If the resulting value is on a day that does not exist on the result month then the day part of the result is decreased. For example:

```
print FormatTime("MONTH DAY, YEAR",AddMonth(TimeValue(2000,03,31),1))
```

will print

```
April 30, 2000
```

25.9 ADDRESS(myFunc())

Return the entry point of a function or subroutine. The returned value is to be used solely in a corresponding See ⟨undefined⟩ [ICALL], page ⟨undefined⟩.

The returned value is an integer value that is the internal node number of the compiled code where the function starts. The different node numbers are in complex relation with each other and simple rules can not be applied. In other words playing around with the value returned by the function `ADDRESS` and then using it in an `ICALL` may result interpreter crash raising internal error.

Note that in the argument of the function `ADDRESS` the function name has to include the `()` characters. The function is NOT called by the code when the function `ADDRESS` is used. On the other hand forgetting the opening and closing parentheses will result erroneous value unusable by `ICALL`.

25.10 ADDSECOND

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the seconds by the second argument and returns the time value.

This function is the simplest from the arithmetic's point of view, because it simply adds the second argument to the first argument and returns the result.

25.11 ADDWEEK

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the week by the second argument and returns the time value for the same hour and minute but some weeks later or sooner in case the second argument is negative.

This function is very simple from the arithmetic's point of view, because it simply adds 604800 times the second argument to the first argument and returns the result.

25.12 ADDYEAR

This function takes two arguments. The first argument is a time value, the second is an integer value. The function increments the year of the time value by the second argument and returns the time value for the same month, day, hour and minute but some years later or sooner in case the second argument is negative.

This is a bit more complex than just adding 365*24*60*60 to the value, because leap-years are longer and in case you add several years to the time value you should consider adding these longer years extra days. This is calculated correct in this function.

If the original time value is February 29 on a leap-year and the resulting value is in a year, which is not leap year the function will return February 28.

Note that because of this correction using the function in a loop is not the same as using it once. For example:

```
print AddYear(TimeValue(2000,02,29),4),"\n"
print AddYear(AddYear(TimeValue(2000,02,29),2),2),"\n"
```

will print two different values.

25.13 ASC(string)

Returns the ASCII code of the first character of the argument string.

25.14 ASECANT

This is a planned function to calculate the arcus secant of the argument.

25.15 ASIN

Calculates the arcus sine of the argument, which is the inverse of the function See ⟨undefined⟩ [SIN], page ⟨undefined⟩. If the argument is not between (-1.0,+1.0) the return value is `undef`.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`ASIN(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.16 ATAN

This is a planned function to calculate the arcus tangent of the argument.

25.17 ATN

This is a planned function to calculate the arcus tangent of the argument.

25.18 BIN

This is a planned function to convert the argument number to binary format. (aka. format as a binary number containing only 0 and 1 characters and return this string)

25.19 BINMODE [# fn] | input | output

Set an opened file handling to binary mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also See ⟨undefined⟩ [TEXTMODE], page ⟨undefined⟩

Set an opened file handling to binary mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also See ⟨undefined⟩ [TEXTMODE], page ⟨undefined⟩

Set an opened file handling to binary mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also See ⟨undefined⟩ [TEXTMODE], page ⟨undefined⟩

25.20 CALL subroutine

Use this command to call a subroutine. Subroutines can be called just writing the name of the already defined subroutine and the arguments. However in situation when the code calls a function that has not yet been defined the interpreter knows that the command is a subroutine call from the keyword `CALL`.

To be safe you can use the keyword before any subroutine call even if the subroutine is already defined.

Subroutines and functions can be called the same way. ScriptBasic does not make real distinction between subroutines and functions. However it is recommended that functions be used as functions using the return value and code segments not returning any value are implemented and called as subroutine.

25.21 CHDIR directory

Change the current working directory (CWD). This command accepts one argument, the directory which has to be the CWD after the command is executed. If the CWD can not be changed to that directory then an error is raised.

Pay careful attention when you use this command in your code. Note that there is only one CWD for each process and not one for each thread. When an application embeds the BASIC interpreter in a multi-thread environment, like in the Eszter SB Application Engine this command may alter the CWD for all the threads.

For this reason the Eszter SB Application Engine switches off this command, raising error if ever a program executed in the engine calls this command whatever argument is given.

Thus usually BASIC programs should avoid calling this command unless the programmer is certain that the BASIC program will only be executed in a single thread environment (command line).

Change the current working directory (CWD). This command accepts one argument, the directory which has to be the CWD after the command is executed. If the CWD can not be changed to that directory then an error is raised.

Pay careful attention when you use this command in your code. Note that there is only one CWD for each process and not one for each thread. When an application embeds the BASIC interpreter in a multi-thread environment, like in the Eszter SB Application Engine this command may alter the CWD for all the threads.

For this reason the Eszter SB Application Engine switches off this command, raising error if ever a program executed in the engine calls this command whatever argument is given.

Thus usually BASIC programs should avoid calling this command unless the programmer is certain that the BASIC program will only be executed in a single thread environment (command line).

Change the current working directory (CWD). This command accepts one argument, the directory which has to be the CWD after the command is executed. If the CWD can not be changed to that directory then an error is raised.

Pay careful attention when you use this command in your code. Note that there is only one CWD for each process and not one for each thread. When an application embeds the BASIC interpreter in a multi-thread environment, like in the Eszter SB Application Engine this command may alter the CWD for all the threads.

For this reason the Eszter SB Application Engine switches off this command, raising error if ever a program executed in the engine calls this command whatever argument is given.

Thus usually BASIC programs should avoid calling this command unless the programmer is certain that the BASIC program will only be executed in a single thread environment (command line).

25.22 CHOMP()

Remove the trailing new line from the space. If the last character of the string is not new line then the original stringis returned. This function is useful to remove the trailing new line character when reading a line from a file using the command See ⟨undefined⟩ [LINEINPUT], page ⟨undefined⟩

25.23 CHR(code)

Return a one character string containing a character of ASCII code `code`.

25.24 CINT

This function is the same as See ⟨undefined⟩ [INT], page ⟨undefined⟩ and is present in ScriptBasic to be more compatible with other BASIC language variants.

25.25 CLOSE [#] fn

Close a previously successfully opened file. The argument of the command is the file number that was used in the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ to open the file.

If the file number is not associated with a successfully opened file then error is raised.

```
REM open the file to read
open "test.bas" for input as 1
REM close the file
close#1

REM open two files for reading
open "test.bas" for input as 1
open "test.sb" for input as 2

REM close all files
close
```

Close a previously successfully opened file. The argument of the command is the file number that was used in the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ to open the file.

If the file number is not associated with a successfully opened file then error is raised.

You can also use the command without any argument. In this case all currently opened files and sockets are going to be closed. For those, who want to express this behaviour this command can be used with the keyword `CLOSEALL`. Note however that the keyword `CLOSEALL` is not a replacement for the keyword `CLOSE`. You can not close a single file or socket using the keyword `CLOSEALL`.

Close a previously successfully opened file. The argument of the command is the file number that was used in the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ to open the file.

If the file number is not associated with a successfully opened file then error is raised.

25.26 CLOSE DIRECTORY [#] dn

Close an opened directory and release all memory that was used by the file list.

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

Close an opened directory and release all memory that was used by the file list.

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

Close an opened directory and release all memory that was used by the file list.

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

25.27 COMMAND()

This function returns the command line arguments of the program in a single string. This does not include the name of the interpreter and the name of the BASIC program, only the arguments that are to be passed to the BASIC program. For example the program started as

```
# scriba test_command.sb arg1 arg2  arg3
```

will see `"arg1 arg2 arg3"` string as the return value of the function `COMMAND()`.

25.27.1 COMMANDF Details

You can start the ScriptBasic interpreter several ways. The most evident way is to specify the interpreter on the command line:

```
# scriba test_command.sb arg1 arg2  arg3
```

You can also start the program using the name of the program:

```
# test_command.sb arg1 arg2  arg3
```

This assumes that the script is executable and that the first line is `#!/usr/bin/scriba` pointing to the interpreter (UNIX). On Windows the file extension `.sb` should be associated with the interpreter. This is done by the ScriptBasic installer. (Also see the note below.)

Under Windows you can also start the code without specifying the extension:

```
# test_command arg1 arg2  arg3
```

You can have the program compiled to executable and you can start the exe with the command

```
# test_command arg1 arg2  arg3
```

where `test_command` now refers to the executable file. In any of these cases the function `COMMAND` returns the string containning the commal line arguments.

NOTE:

Some versions of the installer has a bug that will prevent passing command-line arguments to a BASIC program when it is started on the command by its name. To check whether your have ScriptBasic corretcly installed try execute the following program:

```
REM test_command.sb
print COMMAND()
```

Save the program into the file `test_command.sb` and type the command:

```
C:\> test_command arg1 arg2  arg3
```

If the program prints the arguments, then you are not exposed to this bug. If the program returns without printing anything but a new line you have installed a ScriptBasic version that contains this installer bug.

To correct this bug you should start the registry editor and open the key

```
HKEY_CLASSES_ROOT\ScriptBasic\shell\command
```

The default value should be

```
C:\ScriptBasic\BIN\scriba.exe %1
```

The actual path to the executable may differ from the listed above. If that is the case you can modify the value to be

```
C:\ScriptBasic\BIN\scriba.exe %1 %*
```

You should NOT use the value from this document as listed above, but rather you have to edit the value and append `%*` after the original value, because it is likely that you have a differet installation path of the interpreter.

25.28 Concatenate operator &

This operator concatenates two strings. The resulting string will contain the characters of the string standing on the left side of the operator followed by the characters of the string standing on the right hand side of the operator. The ScriptBasic interpreter automatically allocates the resulting string.

25.29 CONF("conf.key")

This function can be used to retrieve ScriptBasic configuration parameters. This is rarely needed by general programmers. This is needed only for scripts that maintain the ScriptBasic setup, for example install a new module copying the files to the appropriate location.

The argument `"conf.key"` should be the configuration key string. If this key is not on the top level then the levels should be separated using the dot chatacter, like `conf("preproc.internal.dbg")` to get the debugger DLL or SO file.

The return value of the function is the integer, real or string value of the configuration value. If the key is not defined or if the system manager set the key to be hidden (see later) then the function will raise an error

`(0): error &H8:The argument passed to a module function is out of the accepted range.`

Some of the configuration values are not meant to be readable for the BASIC programs for security reasons. A typical example is the database connection password. The system manager can insert extra "dummy" configuration keys that will prevent the BASIC program to get the actual value of the configuration key. The extra configuration key has to have the same name as the key to be hidden with a `$` sign prepended to it.

For example the MySQL connection named `test` has the connection password under the key `mysql.connections.test.password`. If the key in the compiled configuration file `mysql.connections.test.$password` exists then the BASIC function `conf()` will result error. The value of this extra key is not taken into account.

The system manager can configure whole configuration branches to be hidden from the BASIC programs. For example the configuration key `mysql.connections.$test` defined with any value will prevent access of BASIC programs to any argument of the connection named `test`. Similarly the key `mysql.$connections` will prevent access to any configuration value of any MySQL connections if defined and finally the key `$mysql` will stop BASIC programs to discover any MySQL configuration information if defined.

The current implementation does not examine the actual value of the extra security key. However later implementations may alter the behaviour of this function based on the value of the key. To remain compatible with later versions it is recommended that the extra security key is configured to have the value `1`.

25.30 COS

Calculates the cosine of the argument.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`COS(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.31 COSECANT

This is a planned function to calculate the cosecant of the argument.

25.32 COTAN

This is a planned function to calculate the cotangent of the argument.

25.33 COTAN2

This is a planned function to calculate the cotangent of the ratio of the two arguments.

25.34 CRYPT(string,salt)

This function returns the encoded DES digest of the string using the salt as it is used to encrypt passwords under UNIX.

Note that only the first 8 characters of the string are taken into account.

This function returns the encoded DES digest of the string using the salt as it is used to encrypt passwords under UNIX.

Note that only the first 8 characters of the string are taken into account.

This function returns the encoded DES digest of the string using the salt as it is used to encrypt passwords under UNIX.

Note that only the first 8 characters of the string are taken into account.

25.35 CURDIR()

=displax CURDIR()

This function does not accept argument and returns the current working directory as a string. =displax CURDIR()

This function does not accept argument and returns the current working directory as a string. =displax CURDIR()

This function does not accept argument and returns the current working directory as a string.

25.36 CVD

This is a planned function to convert the argument string into a real number.

Converts a passed in string "str$" to a double-precision number. The passed string must be eight (8) bytes or longer. If less than 8 bytes long, an error is generated. If more than 8 bytes long, only the first 8 bytes are used.

25.37 CVI

This is a planned function to convert the argument string into an integer.

Converts a passed in string "str$" to an integer number. The passed string must be two (2) bytes or longer. If less than 2 bytes long, an error is generated. If more than 2 bytes long, only the first 2 bytes are used.

25.38 CVL

This is a planned function to convert the argument string into an integer.

Converts a passed in string "str$" to a long-integer number. The passed string must be four (4) bytes or longer. If less than 4 bytes long, an error is generated. If more than 4 bytes long, only the first 4 bytes are used.

25.39 CVS

This is a planned function to convert the argument string into an integer.

Converts a passed in string "str$" to a single precision number. The passed string must be four (4) bytes or longer. If less than 4 bytes long, an error is generated. If more than 4 bytes long, only the first 4 bytes are used.

25.40 DAY

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the day of the month (1 to 31) value of that time. If the argument is missing the function uses the actual local time to calculate the day of the month value. In other words it returns the day value of the actual date.

25.41 DECLARE COMMAND function ALIAS cfun LIB library

This command is used to declare a command implemented in an external ScriptBasic library. Do NOT use this command to invoke a function from an external DLL that was not specifically written for ScriptBasic. When you include module BASIC files that contain `DECLARE COMMAND` lines, you can call the functions declared this way as they were entirely written in BASIC. You use/write a `DECLARE COMMAND` command if you developed an external module for ScriptBasic programs in C.

25.41.1 DECLARECOMMAND Details

Declare a command implemented in an external library. Note that library and the command implemented in it should be specifically written for ScriptBasic. This command is not able to declare and call just any library function.

The arguments are similar to that of the command `DECLARE SUB` and the same restrictions and conditions apply. Look at the documentation of the command See ⟨undefined⟩ [DECLARESUB], page ⟨undefined⟩.

DEVELOPER DETAILS!

This command is used to start an external command defined in a dynamic load library. Please read the details of the command `DECLARE SUB` if you did not do up to know before reading the details of this command.

The major difference between an external function and an external command is that arguments are passed after they are evaluated to external functions, while arguments are not evaluated for external commands.

External commands are a bit more tedious to implement. On the other hand external commands have more freedom handling their arguments. A command is allowed to evaluate or not to evaluate some of its arguments. It can examine the structure of the expression passed as argument and evaluate it partially at its wish.

From the BASIC programmer point of view external commands are called the same way as user defined functions or external functions.

25.42 DECLARE SUB function ALIAS cfun LIB library

This command is used to declare a function implemented in an external ScriptBasic library. Do NOT use this command to invoke a function from an external DLL that was not specifically written for ScriptBasic. When you include module BASIC files that contain `DECLARE SUB` lines, you can call the functions declared this way as they were entirely written in BASIC. You use/write a `DECLARE SUB` command if you developed an external module for ScriptBasic programs in C.

The difference between `DECLARE SUB` and `DECLARE COMMAND` is that the arguments passed to a function declared using `DECLARE SUB` evaluates its argument and passes the values to the C program implementing the function, whereas the functions declared using the command `DECLARE COMMAND` starts the function and evaluate the arguments one-by-one when and if the function implemented in C requests.

This difference is only important when you use expressions in the place of an argument that itself calls some other functions and has so called side effect. Have a look at the following code:

```
import iff.bas

function side_effect()
  b = 1 + b
  side_effect = b
end function

b = 0
print iff(0,side_effect(),2)
print b
```

In the example above we use a hipotethical function implemented by a module and declared in the file `iff.bas`. This function evaluates the first argument and if that is true returns the second argument, otherwise it returns the third argument.

If the function `iff` was implemented as a command and declared accordingly using the command `DECLARE COMMAND` and if that module function evaluates only one of the second and third arguments then the global variable `b` remains unchanged.

If the function `iff` was implemented as a function and declared accordingly using the command `DECLARE SUB` and then the global variable `b` is increased.

25.42.1 DECLARESUB Details

Declare a function implemented in an external library. Note that library and the function implemented in it should be specifically written for ScriptBasic. This command is not able to declare and call just any library function.

The command has three arguments. The argument `function` is a symbol (a case insensitive string without any space, containing only alphanumeric characters and without double quotes) , the name of the function as it is going to be used in the BASIC program. The BASIC programmer is free to choose this function name. The function later can be used as if it was a BASIC user defined function.

The argument `cfun` has to be the name of the function as it is defined in the library. This has to be a constant string value. You can not use any expression here only a constant string. This is usually the same as the name of the interface function in the C source file. The programmer writing the module in C has to know this name. If for some reason you do not happen to know it, but you need to declare the function you may be lucky using the name that stands in the C source file between parentheses after the macro `besFUNCTION`.

The last argument is the name of the library. This also has to be a constant string value. You can not use any expression here but a constant string. This argument has to specify the name of the library file without the extension and the path where the library is located. The extension will automatically be appended to the file name and the path will automatically be prepended to it. The actual extension and the path to be searched for the library is defined in the ScriptBasic configuration file.

You can also specify the full file name containing the full path to the library as well as the file extension. In this case the ScriptBasic configuration file data for the module path and extension is ignored.

DEVELOPER DETAILS!

This command is used to start an external function defined in a dynamic load library.

The dynamically loaded modules are implemented in ScriptBasic via an ordinary command that has the syntax:

```
'declare' 'sub' * function 'alias' string 'lib' string nl
```

For the compiler it generates a user defined function, which is defined on the line that contains the declare statement. The execution system will call itself recursively to execute lines starting at the line where the `declare` statement is. The command implemented in this file is executed and unlike the `FUNCTION` or `SUB` it immediately tells the execution system to return after the line was executed.

This command first checks if the line was already executed. On the first execution it loads the module and gets the address of the function named in the alias string. This entry point is stored in a `struct` and next time the function is called by the basic pointer it does not need to search for the function and for the module. If a function of an already loaded module is called the program does not reload the module. The program maintains a linked list with the names of the loaded modules and loads modules only when they are first referenced.

The modules are loaded using the operating system `dll` loading function `dlopen` or `LoadLibrary`. These functions search several locations for libraries in case the library is specified without absolute path name.

The module loader can be fouled if the same library is defined with full path and with single name in the basic code.

For example, if the commands

```
declare sub fun2 alias "mefun" lib "libobas"
declare sub fun3 alias "youfun" lib "/usr/lib/scriba/libobas.so"
```

refer to the same module, the code implemented here thinks that they are different libraries.

When the module is loaded the code tries to get the function named `versmodu` and calls it. This function gets three arguments. The first argument is the interface version that ScriptBasic supports for the external modules. The second argument is a pointer to a ZCHAR terminated string that contains the variation of the calling interpreter. Note that this has to be an 8-character (+1 ZCHAR) string. The third argument is a pointer to a `NULL` initialized `void *` pointer, which is the module pointer. This pointer is stored by ScriptBasic, and it is guaranteed not been modified. The modules can store "global" variable information using this pointer. Usually this pointer is not used in this function, especially because there are no "safe" memory allocation functions available at this point of execution.

The module should examine the version it gets and it should decide if the interface version is OK for the module. If the interface version is not known the function should return 0 and ScriptBasic will interpret this value as a failure to load the module. If the module does not know if the interface version is good or not it can return the interface version that it can handle. In this case it is the duty of the interpreter to decide if the interface version can be provided or not. ScriptBasic will examine the version and in case it can not handle the version it will generate an error.

This methodology will allow either ScriptBasic to revert its functionality to earlier interface versions in case the higher version interface not only extends the lower version but is incompatible with the former version. On the other hand a module designed for a higher version may be loaded and executed by a lower version of ScriptBasic in case the module is ready to use the lower interface version.

If the function `versmodu` can not be found in the DLL then ScriptBasic assumes that the module is ready to accept the current interface version. However such a module should not generally be written.

Note that the version we are talking about here is neither the version of ScriptBasic nor the version of the module. This is the version of the interface between the module and ScriptBasic.

After the version negotiation has been successfully done ScriptBasic tries to get the address of the function named `bootmodu`. If this function exists it is started. This function can perform all the initializations needed for the module. This function gets all the parameters that a usual module implemented function except that there are no parameters and the function should not try to set a result value, because both of these arguments are `NULL`.

A module function (including `bootmodu`) is called with four arguments. The four arguments are four pointers.

```
int my_module_function(pSupportTable pSt,
                       void **ppModuleInternal,
```

```
                                pFixSizeMemoryObject pParameters,   // NULL for bootmodu
                                pFixSizeMemoryObject *pReturnValue) // NULL for bootmodu
```

The parameter `pSt` points to a `struct` that holds the function pointers that the function can call to reach ScriptBasic functionality. These functions are needed to access the arguments and to return a value.

The parameter `ppModuleInternal` is a pointer pointing to a `NULL` initialized pointer. This pointer belongs to the module. ScriptBasic guarantees that the value is not modified during the execution (unless the module itself modifies it). This pointer can be used to remember the address space allocated by the module for itself. To store permanent values to remember the state of the module you can either use static variables or this pointer. However using this pointer and allocating memory to store the values is the preferred method. The reason for preferring this method is that global variables are global in the whole process. On the other hand ScriptBasic may run in several threads in a single process executing several different basic programs. This `ppModuleInternal` pointer will be unique for each thread.

Here is the point to discuss a bit the role of `bootmodu`. Why not to use `DllMain` under Windows NT? The reason is in the possibility of threaded execution. `DllMain` is executed when the process loads the `dll`. `bootmodu` is executed when the basic executor loads the module. If there are more threads running then `DllMain` is not started again. Use `DllMain` or a similar function under UNIX if you want to initialize some process level data. Use `bootmodu` to initialize basic program specific data.

The parameter `pParameters` is a ScriptBasic array containing the values of the arguments. There is no run-time checking about the number of arguments. This is up to the module function.

The last parameter is a pointer to a ScriptBasic variable. The initial value of the pointed pointer is `NULL`, meaning `undef` return value. The final return value should be allocated using the macros defined in `basext.h` and this pointer should be set to point to the value. Note however that `bootmodu` and `finimodu` should not to try to dereference this variable, because for both of them the value is `NULL`.

For further information on how to write module extension functions read the on-line documentation of `basext.c` in this source documentation.

A module is unloaded when the basic program execution is finished. There is no basic code to unload a module. (Why?)

Before the module is unloaded calling one of the operating system functions `dlclose` or `FreeLibrary` the program calls the module function `finimodu` if it exists. This function gets all the four pointers (the last two are `NULL`s) and it can perform all the tasks that the module has to do clean up.

Note however that the module need not release the memory it allocated using the `besALLOCATE` macro defined in the file `basext.h` (which is generated using `headerer.pl` from `basext.c`). The memory is going to be released afterwards by ScriptBasic automatically.

You can have a look at the source code of modules provided by ScriptBasic, for example MySQL module, Berkeley DB module, HASH module, MT module and so on.

25.43 DELETE file/directory_name

This command deletes a file or an **empty>** directory. You can not delete a directory which contains files or subdirectories.

If the file or the directory can not be deleted an error is raised. This may happen for example if the program trying to delete the file or directory does not have enough permission.

See See ⟨undefined⟩ [DELTREE], page ⟨undefined⟩ for a more powerful and dangerous delete.

This command deletes a file or an **empty>** directory. You can not delete a directory which contains files or subdirectories.

If the file or the directory can not be deleted an error is raised. This may happen for example if the program trying to delete the file or directory does not have enough permission.

See See ⟨undefined⟩ [DELTREE], page ⟨undefined⟩ for a more powerful and dangerous delete.

This command deletes a file or an **empty>** directory. You can not delete a directory which contains files or subdirectories.

If the file or the directory can not be deleted an error is raised. This may happen for example if the program trying to delete the file or directory does not have enough permission.

See See ⟨undefined⟩ [DELTREE], page ⟨undefined⟩ for a more powerful and dangerous delete.

25.44 DELTREE file/directory_name

Delete a file or a directory. You can use this command to delete a file the same way as you do use the command See ⟨undefined⟩ [DELETE], page ⟨undefined⟩. The difference between the two commands See ⟨undefined⟩ [DLETE], page ⟨undefined⟩ and `DELTREE` comes into place when the program deletes directories.

This command, `DELTREE` forcefully tries to delete a directory even if the directory is not empty. If the directory is not empty then the command tries to delete the files in the directory and the subdirectories recursively.

If the file or the directory cannot be deleted then the command raises error. However even in this case some of the files and subdirectories may already been deleted.

Delete a file or a directory. You can use this command to delete a file the same way as you do use the command See ⟨undefined⟩ [DELETE], page ⟨undefined⟩. The difference between the two commands See ⟨undefined⟩ [DLETE], page ⟨undefined⟩ and `DELTREE` comes into place when the program deletes directories.

This command, `DELTREE` forcefully tries to delete a directory even if the directory is not empty. If the directory is not empty then the command tries to delete the files in the directory and the subdirectories recursively.

If the file or the directory cannot be deleted then the command raises error. However even in this case some of the files and subdirectories may already been deleted.

Delete a file or a directory. You can use this command to delete a file the same way as you do use the command See ⟨undefined⟩ [DELETE], page ⟨undefined⟩. The difference

between the two commands See ⟨undefined⟩ [DLETE], page ⟨undefined⟩ and `DELTREE` comes into place when the program deletes directories.

This command, `DELTREE` forcefully tries to delete a directory even if the directory is not empty. If the directory is not empty then the command tries to delete the files in the directory and the subdirectories recursively.

If the file or the directory cannot be deleted then the command raises error. However even in this case some of the files and subdirectories may already been deleted.

25.45 DO

This command is a looping construct that repeats commands so long as long the condition following the keyword `UNTIL` becomes `true` or the condition following the keyword `WHILE` becomes `false`.

The format of the command is

```
DO
 ...
 commands to repeat
 ...
LOOP WHILE expression
```

or

```
DO
 ...
 commands to repeat
 ...
LOOP UNTIL expression
```

The condition expression is evaluated every time after the loop commands were executed. This means that the loop body is executed at least once.

A `DO`/`LOOP` construct should be closed with a `LOOP UNTIL` or with a `LOOP WHILE` command but not with both.

The `DO`/`LOOP UNTIL` is practically equivalent to the `REPEAT`/`UNTIL` construct.

See also See ⟨undefined⟩ [WHILE], page ⟨undefined⟩, See ⟨undefined⟩ [DOUNTIL], page ⟨undefined⟩, See ⟨undefined⟩ [DOWHILE], page ⟨undefined⟩, See ⟨undefined⟩ [REPEAT], page ⟨undefined⟩, See ⟨undefined⟩ [DO], page ⟨undefined⟩ and See ⟨undefined⟩ [FOR], page ⟨undefined⟩.

25.46 DO UNTIL condition

This command implements a looping construct that loops the code between the line `DO UNTIL` and `LOOP` util the expression following the keywords on the loop starting line becomes `true`.

```
DO UNTIL expression
 ...
 commands to repeat
```

```
 ...
LOOP
```

The expression is evaluated when the looping starts and each time the loop is restarted. It means that the code between the `DO UNTIL` and `LOOP` lines may be skipped totally if the expression evaluates to some `TRUE` value during the first evaluation before the execution starts the loop.

This command is practically equivalent to the construct

```
WHILE NOT expression
...
 commands to repeat
 ...
WEND
```

You can and you also should use the construct that creates more readable code.

See also See ⟨undefined⟩ [WHILE], page ⟨undefined⟩, See ⟨undefined⟩ [DOUNTIL], page ⟨undefined⟩, See ⟨undefined⟩ [DOWHILE], page ⟨undefined⟩, See ⟨undefined⟩ [REPEAT], page ⟨undefined⟩, See ⟨undefined⟩ [DO], page ⟨undefined⟩ and See ⟨undefined⟩ [FOR], page ⟨undefined⟩.

25.47 DO WHILE condition

This command implements a looping construct that loops the code between the line `DO WHILE` and `LOOP` util the expression following the keywords on the loop starting line becomes `false`.

Practically this command is same as the command See ⟨undefined⟩ [WHILE], page ⟨undefined⟩ with a different syntax to be compatible with different BASIC implementations.

```
do while
 ...
loop
```

You can use the construct that creates more readable code.

See also See ⟨undefined⟩ [WHILE], page ⟨undefined⟩, See ⟨undefined⟩ [DOUNTIL], page ⟨undefined⟩, See ⟨undefined⟩ [DOWHILE], page ⟨undefined⟩, See ⟨undefined⟩ [REPEAT], page ⟨undefined⟩, See ⟨undefined⟩ [DO], page ⟨undefined⟩ and See ⟨undefined⟩ [FOR], page ⟨undefined⟩.

25.48 END

End of the program. Stops program execution.

You should usually use this command to signal the end of a program. Although you can use See ⟨undefined⟩ [STOP], page ⟨undefined⟩ and `END` interchangeably this is the convention in BASIC programs to use the command `END` to note the end of the program and `STOP` to stop the program execution based on some extra condition inside the code.

25.49 ENVIRON("envsymbol") or ENVIRON(n)

This function returns the value of an environment variable. Environment variables are string values associated to names that are provided by the executing environment for the programs. The executing environment is usually the operating system, but it can also be the Web server in CGI programs that alters the environment variables provided by the surrounding operating system specifying extra values.

This function can be used to get the string of an environment variable in case the program knows the name of the variable or to list all the environment variables one by one.

If the environment variable name is known then the name as a string has to be passed to this function as argument. In this case the return value is the value of the environment variable. For example

```
MyPath = ENVIRON("PATH")
```

If the program wants to list all the environment variables the argument to the function `ENVIRON` should be an integer number `n`. In this case the function returns a string containing the name and the value of the `n`-th environment variable joined by a `=` sign. The numbering starts with `n=0`.

If the argument value is integer and is out of the range of the possible environment variable ordinal numbers (negative or larger or equal to the number of the available environment variables) then the function returns `undef`.

If the argument to the function is `undef` then the function also returns the `undef` value.

Note that ScriptBasic provides an easy way for the embedding applications to redefine the underlying function that returns the environment variable. Thus an embedding application can "fool" a BASIC program providing its own environment variable. For example the Eszter SB Application Engine provides an alternative environment variable reading function and thus BASIC applications can read the environment using the function `ENVIRON` as if the program was running in a real CGI environment.

25.49.1 ENVIRON Details

The following sample code prints all environment variables:

```
i=0
do
  e$ = environ(i)
  if IsDefined(e$) then
    print e$
    print
  endif
  i = i + 1
loop while IsDefined(e$)
```

25.50 EOD(dn)

Checks if there is still some file names in the directory opened for reading using the directory number `dn`.

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

Checks if there is still some file names in the directory opened for reading using the directory number `dn`.

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

Checks if there is still some file names in the directory opened for reading using the directory number `dn`.

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

25.51 EOF(n)

This function accepts one parameter, an opened file number. The return value is `true` if and only if the reading has reached the end of the file.

This function accepts one parameter, an opened file number. The return value is `true` if and only if the reading has reached the end of the file.

This function accepts one parameter, an opened file number. The return value is `true` if and only if the reading has reached the end of the file.

25.52 ERROR() or ERROR n

The keyword `ERROR` can be used as a command as well as a built-in function. When used as a function it returns the error code that last happened. The error codes are defined in the header file `error.bas` that can be included with the command `import`. The error code is a vital value when an error happens and is captured by some code defined after the label referenced by the command `ON ERROR GOTO`. This helps the programmer to ensure that the error was really the one that the error handling code can handle and not something else.

If the keyword is used as a command then it has to be followed by some numeric value. The command does not ever return but generates an error of the code given by the argument.

Take care when composing the expression following the keyword `ERROR`. It may happen that the expression itself can not be evaluated due to some error conditions during the evaluation of the expression. The best practice is to use a constant expression using the symbolic constants defined in the include file `error.bas`.

Note that the codes defined in the include file are version dependant.

If an error is not captured by any `ON ERROR GOTO` specified error handler then the interpreter stops. The command line variation passes the error code to the executing environment as exit code. In other word you can exit from a BASIC program

25.53 ERROR$() or ERROR$(n)

Returns the English sentence describing the last error if the argument is not defined or returns the English sentence describing the error for the error code `n`.

If the error code `n` provided as argument is negative or is above all possible errors then the result of the function is `undef`.

25.54 EVEN

Return `true` if the argument is an even number. `EVEN(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

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

25.55 EXECUTE("executable_program", time_out,pid_v)

This function should be used to start an external program and wait for it to finish.

The first argument of the function is the executable command line to start. The second argument is the number of seconds that the BASIC program should wait for the external program to finish. If the external program finishes during this period the function returns and the return value is the exit code of the external program. If the argument specifying how many seconds the BASIC program has to wait is `-1` then the BASIC program will wait infinitely.

If the program does not finish during the specified period then the function alters the third argument, which has to be a variable and raises error. In this case the argument `pid_v` will hold the PID of the external program. This value can be used in the error handling code to terminate the external program.

25.55.1 EXECUTE Details

The function can be used to start a program synchronous and asynchronous mode as well. When the timeout value passed to the function is zero the function starts the new process, but does not wait it to finish, but raises an error. In this case the BASIC program can catch this error using the `ON ERROR GOTO` structure and get the pid of the started process from the variable `pid_v`. In this case the function does not "return" any value because a BASIC error happened. For example:

```
ON ERROR GOTO NoError
a = EXECUTE("ls",0,PID)
NoError:
print "The program 'ls' is running under the pid: ",PID,"\n"
```

If the argument `time_out` is `-1` the function will wait for the subprocess to finish whatever long it takes to run. For example:

```
a = EXECUTE("ls",-1,PID)
print "ls was executed and the exit code was:",a
```

Note that the string passed as first argument containing the executable program name and the arguments (the command line) should not contain zero character (a character with ASCII code 0) for security reasons. If the command line string contains zero character an error is raised.

This function should be used to start an external program and wait for it to finish.

The first argument of the function is the executable command line to start. The second argument is the number of seconds that the BASIC program should wait for the external program to finish. If the external program finishes during this period the function returns

and the return value is the exit code of the external program. If the argument specifying how many seconds the BASIC program has to wait is `-1` then the BASIC program will wait infinitely.

If the program does not finish during the specified period then the function alters the third argument, which has to be a variable and raises error. In this case the argument `pid_v` will hold the PID of the external program. This value can be used in the error handling code to terminate the external program.

25.55.2 EXECUTE Details

The function can be used to start a program synchronous and asynchronous mode as well. When the timeout value passed to the function is zero the function starts the new process, but does not wait it to finish, but raises an error. In this case the BASIC program can catch this error using the `ON ERROR GOTO` structure and get the pid of the started process from the variable `pid_v`. In this case the function does not "return" any value because a BASIC error happened. For example:

```
ON ERROR GOTO NoError
a = EXECUTE("ls",0,PID)
NoError:
print "The program 'ls' is running under the pid: ",PID,"\n"
```

If the argument `time_out` is `-1` the function will wait for the subprocess to finish whatever long it takes to run. For example:

```
a = EXECUTE("ls",-1,PID)
print "ls was executed and the exit code was:",a
```

Note that the string passed as first argument containing the executable program name and the arguments (the command line) should not contain zero character (a character with ASCII code 0) for security reasons. If the command line string contains zero character an error is raised.

This function should be used to start an external program and wait for it to finish.

The first argument of the function is the executable command line to start. The second argument is the number of seconds that the BASIC program should wait for the external program to finish. If the external program finishes during this period the function returns and the return value is the exit code of the external program. If the argument specifying how many seconds the BASIC program has to wait is `-1` then the BASIC program will wait infinitely.

If the program does not finish during the specified period then the function alters the third argument, which has to be a variable and raises error. In this case the argument `pid_v` will hold the PID of the external program. This value can be used in the error handling code to terminate the external program.

25.55.3 EXECUTE Details

The function can be used to start a program synchronous and asynchronous mode as well. When the timeout value passed to the function is zero the function starts the new process, but does not wait it to finish, but raises an error. In this case the BASIC program

can catch this error using the `ON ERROR GOTO` structure and get the pid of the started process from the variable `pid_v`. In this case the function does not "return" any value because a BASIC error happened. For example:

```
ON ERROR GOTO NoError
a = EXECUTE("ls",0,PID)
NoError:
print "The program 'ls' is running under the pid: ",PID,"\n"
```

If the argument `time_out` is `-1` the function will wait for the subprocess to finish whatever long it takes to run. For example:

```
a = EXECUTE("ls",-1,PID)
print "ls was executed and the exit code was:",a
```

Note that the string passed as first argument containing the executable program name and the arguments (the command line) should not contain zero character (a character with ASCII code 0) for security reasons. If the command line string contains zero character an error is raised.

25.56 EXIT FUNCTION

This function stops the execution of a function and the execution gets back to the point from where the function was called. Executing this command has the same effect as if the execution has reached the end of a function.

25.57 EXIT SUB

This function stops the execution of a subroutine and the execution gets back to the point from where the subroutine was called. Executing this command has the same effect as if the execution has reached the end of a subroutine.

Same as See ⟨undefined⟩ [EXITFUNC], page ⟨undefined⟩

25.58 EXP

Calculates the x-th exponent of `e`. If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`EXP(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.59 FALSE

This built-in constant is implemented as an argument less function. Returns the value `false`.

25.60 FILEACCESSTIME(file_name)

Get the time the file was accessed last time.

This file time is measured in number of seconds since January 1, 1970 00:00. Note that the different file systems store the file time with different precision. Also FAT stores the file time in local time while NTFS for example stores the file time as GMT. This function returns the value rounded to whole seconds as returned by the operating system. Some of the file systems may not store all three file time types:

the time when the file was created,

last time the file was modified and

last time the file was accessed

Trying to get a time not defined by the file system will result `undef`.

Get the time the file was accessed last time.

This file time is measured in number of seconds since January 1, 1970 00:00. Note that the different file systems store the file time with different precision. Also FAT stores the file time in local time while NTFS for example stores the file time as GMT. This function returns the value rounded to whole seconds as returned by the operating system. Some of the file systems may not store all three file time types:

the time when the file was created,

last time the file was modified and

last time the file was accessed

Trying to get a time not defined by the file system will result `undef`.

Get the time the file was accessed last time.

This file time is measured in number of seconds since January 1, 1970 00:00. Note that the different file systems store the file time with different precision. Also FAT stores the file time in local time while NTFS for example stores the file time as GMT. This function returns the value rounded to whole seconds as returned by the operating system. Some of the file systems may not store all three file time types:

the time when the file was created,

last time the file was modified and

last time the file was accessed

Trying to get a time not defined by the file system will result `undef`.

25.61 FILECOPY filename,filename

Copy a file. The first file is the existing one, the second is the name of the new file. If the destination file already exists then the command overwrites the file. If the destination file is to be created in a directory that does not exist yet then the directory is created automatically.

In the current version of the command you can not use wild characters to specify more than one file to copy, and you can not concatenate files using this command. You also have

to specify the full file name as destination file and this is an error to specify only a directory where to copy the file.

Later versions of this command may implement these features.

If the program can not open the source file to read or the destination file can not be created then the command raises error.

Copy a file. The first file is the existing one, the second is the name of the new file. If the destination file already exists then the command overwrites the file. If the destination file is to be created in a directory that does not exist yet then the directory is created automatically.

In the current version of the command you can not use wild characters to specify more than one file to copy, and you can not concatenate files using this command. You also have to specify the full file name as destination file and this is an error to specify only a directory where to copy the file.

Later versions of this command may implement these features.

If the program can not open the source file to read or the destination file can not be created then the command raises error.

Copy a file. The first file is the existing one, the second is the name of the new file. If the destination file already exists then the command overwrites the file. If the destination file is to be created in a directory that does not exist yet then the directory is created automatically.

In the current version of the command you can not use wild characters to specify more than one file to copy, and you can not concatenate files using this command. You also have to specify the full file name as destination file and this is an error to specify only a directory where to copy the file.

Later versions of this command may implement these features.

If the program can not open the source file to read or the destination file can not be created then the command raises error.

25.62 FILECREATETIME(file_name)

Get the time the file was modified last time. See also the comments on the function See ⟨undefined⟩ [FTACCESS], page ⟨undefined⟩. Get the time the file was modified last time. See also the comments on the function See ⟨undefined⟩ [FTACCESS], page ⟨undefined⟩. Get the time the file was modified last time. See also the comments on the function See ⟨undefined⟩ [FTACCESS], page ⟨undefined⟩.

25.63 FILEEXISTS(file_name)

Returns `true` if the named file exists. Returns `true` if the named file exists. Returns `true` if the named file exists.

25.64 FILELEN(file_name)

Get the length of a named file. If the length of the file can not be determined (for example the file does not exists, or the process running the code does not have permission to read the file) then the return value is `undef`.

This function can be used instead of See ⟨undefined⟩ [LOC], page ⟨undefined⟩ when the file is not opened by the BASIC program. Get the length of a named file. If the length of the file can not be determined (for example the file does not exists, or the process running the code does not have permission to read the file) then the return value is `undef`.

This function can be used instead of See ⟨undefined⟩ [LOC], page ⟨undefined⟩ when the file is not opened by the BASIC program. Get the length of a named file. If the length of the file can not be determined (for example the file does not exists, or the process running the code does not have permission to read the file) then the return value is `undef`.

This function can be used instead of See ⟨undefined⟩ [LOC], page ⟨undefined⟩ when the file is not opened by the BASIC program.

25.65 FILEMODIFYTIME(file_name)

Get the time the file was modified last time. See also the comments on the function See ⟨undefined⟩ [FTACCESS], page ⟨undefined⟩.

Get the time the file was modified last time. See also the comments on the function See ⟨undefined⟩ [FTACCESS], page ⟨undefined⟩.

Get the time the file was modified last time. See also the comments on the function See ⟨undefined⟩ [FTACCESS], page ⟨undefined⟩.

25.66 FIX

This function returns the integral part of the argument. The return value of the function is integer with the exception that `FIX(undef)` may return `undef`.

`FIX(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

The difference between `INT` and `FIX` is that `INT` truncates down while `FIX` truncates towards zero. The two functions are identical for positive numbers. In case of negative arguments `INT` will give a smaller number if the argument is not integer. For example:

```
int(-3.3) = -4
fix(-3.3) = -3
```

See See ⟨undefined⟩ [INT], page ⟨undefined⟩.

25.67 LOCK # fn, mode

Lock a file or release a lock on a file. The `mode` parameter can be `read`, `write` or `release`.

When a file is locked to `read` no other program is allowed to write the file. This ensures that the program reading the file gets consistent data from the file. If a program locks a

file to read using the lock value `read` other programs may also get the `read` lock, but no program can get the See ⟨undefined⟩ [write], page ⟨undefined⟩ lock. This means that any program trying to write the file and issuing the command `LOCK` with the parameter `write` will stop and wait until all read locks are released.

When a program write locks a file no other program can read the file or write the file.

Note that the different operating systems and therefore ScriptBasic running on different operating systems implement file lock in different ways. UNIX operating systems implement so called advisory locking, while Windows NT implements mandatory lock.

This means that a program under UNIX can write a file while another program has a read or write lock on the file if the other program is not good behaving and does not ask for a write lock. Therefore this command under UNIX does not guarantee that any other program is not accessing the file simultaneously.

Contrary Windows NT does lock the file in a hard way, and this means that no other process can access the file in prohibited way while the file is locked.

This different behavior usually does not make harm, but in some rare cases knowing it may help in debugging some problems. Generally you should not have a headache because of this.

You should use this command to synchronize the BASIC programs running parallel and accessing the same file.

You can also use the command `LOCK REGION` to lock a part of the file while leaving other parts of the file accessible to other programs.

If you heavily use record oriented files and file locks you may consider using some data base module to store the data in database instead of plain files. Lock a file or release a lock on a file. The `mode` parameter can be `read`, `write` or `release`.

When a file is locked to `read` no other program is allowed to write the file. This ensures that the program reading the file gets consistent data from the file. If a program locks a file to read using the lock value `read` other programs may also get the `read` lock, but no program can get the See ⟨undefined⟩ [write], page ⟨undefined⟩ lock. This means that any program trying to write the file and issuing the command `LOCK` with the parameter `write` will stop and wait until all read locks are released.

When a program write locks a file no other program can read the file or write the file.

Note that the different operating systems and therefore ScriptBasic running on different operating systems implement file lock in different ways. UNIX operating systems implement so called advisory locking, while Windows NT implements mandatory lock.

This means that a program under UNIX can write a file while another program has a read or write lock on the file if the other program is not good behaving and does not ask for a write lock. Therefore this command under UNIX does not guarantee that any other program is not accessing the file simultaneously.

Contrary Windows NT does lock the file in a hard way, and this means that no other process can access the file in prohibited way while the file is locked.

This different behavior usually does not make harm, but in some rare cases knowing it may help in debugging some problems. Generally you should not have a headache because of this.

You should use this command to synchronize the BASIC programs running parallel and accessing the same file.

You can also use the command `LOCK REGION` to lock a part of the file while leaving other parts of the file accessible to other programs.

If you heavily use record oriented files and file locks you may consider using some data base module to store the data in database instead of plain files.

25.68 FOR var=exp_start TO exp_stop [STEP exp_step]

Implements a FOR loop. The variable `var` gets the value of the start expression `exp_start`, and after each execution of the loop body it is incremented or decrement by the value `exp_step` until it reaches the stop value `exp_stop`.

```
FOR var= exp_start TO exp_stop [ STEP exp_step]
 ...
 commands to repeat
 ...
NEXT var
```

The `STEP` part of the command is optional. If this part is missing then the default value to increment the variable is 1.

If

the expression `exp_start` is larger than the expression `exp_stop` and `exp_step` is positive or if

the expression `exp_start` is smaller than the expression `exp_stop` and `exp_step` is negative

then the loop body is not executed even once and the variable retains its old value.

When the loop is executed at least once the variable gets the values one after the other and after the loop exists the loop variable holds the last value for which the loop already did not execute. Thus

```
for h= 1 to 3
next
print h
stop
```

prints 4.

The expression `exp_start` is evaluated only once when the loop starts. The other two expressions `exp_stop` and `exp_step` are evaluated before each loop. Thus

```
j = 1
k = 10
for h= 1 to k step j
print h,"\n"
j += 1
k -= 1
next
print k," ",j,"\n"
```

```
stop
```

will print

```
1
3
6
7 4
```

To get into more details the following example loop

```
STEP_v = 5
for z= 1 to 10 step STEP_v
 print z,"\n"
 STEP_v -= 10
next z
```

executes only once. This is because the step value changes its sign during the evaluation and the new value being negative commands the loop to terminate as the loop variable altered value is smaller then the end value. In other words the comparison also depends on the actual value of the step expression.

These are not only the expressions that are evaluated before each loop, but the variable as well. If the loop variable is a simple variable then this has not too much effect. However if the loop variable is an array member then this really has to be taken into account. For example:

```
for j=1 to 9
  A[j] = 0
next

j = 1
for A[j]= 1 to 9

 for k=1 to 9
  print A[k]
 next k
 print

 j += 1
 if j > 9 then STOP

next
```

prints

```
100000000
110000000
111000000
111100000
111110000
111111000
111111100
111111110
```

```
111111111
```

so you can see that the loop takes, evaluates, compares and increments the actual array element as the variable `j` in the sample code above is incremented.

The loop variable or some other left value has to stand between the keyword `FOR` and the sign `=` on the start line of the loop but this is optional following the keyword `NEXT`. ScriptBasic optionally allow you to write the variable name after the keyword `NEXT` but the interpreter does not check if the symbol is a variable of the loop. The role of this symbol is merely documentation of the BASIC code. However, you can not write an array element following the keyword `NEXT`, only a simple variable name.

If the expression `exp_step` is zero then the loop variable is not altered and the loop is re-executed with the same loop variable value. This way you can easily get into infinite loop.

These are fine tuning details of the command `FOR` that you may need to be aware when you read some tricky code. On the other hand you should never create any code that depends on these features. The loop variable is recommended to be a simple variable and the expressions in the loop head should evaluate the same for each execution of the loop. If you need something more special that may depend on some of the features discussed above then you have to consider using some other looping construct to get more readable code.

25.69 FORK()

NOT IMPLEMENTED

This function is supposed to perform process forking just as the native UNIX function `fork` does. However this function is not implemented in ScriptBasic (yet). Until this function is implemented in ScriptBasic you can use the UX module fork function.

This function is supposed to perform process forking just as the native UNIX function `fork` does. However this function is not implemented in ScriptBasic (yet). Until this function is implemented in ScriptBasic you can use the UX module fork function.

This function is supposed to perform process forking just as the native UNIX function `fork` does. However this function is not implemented in ScriptBasic (yet). Until this function is implemented in ScriptBasic you can use the UX module fork function.

25.70 FORMAT()

The function format accepts variable number of arguments. The first argument is a format string and the rest of the arguments are used to create the result string according to the format string. This way the function `format` is like the C function `sprintf`.

The format string can contain normal characters and control substrings.

The control substring have the form `%[flags][width][.precision]type`. It follows the general `sprintf` format except that type prefixes are not required or allowed and type can only be `"dioxXueEfgGsc"`. The `*` for width and precision is supported.

An alternate format BASIC-like for numbers has the form `%~format~` where `format` can be:

Digit or space

0 Digit or zero

^ Stores a number in exponential format. Unlike QB's USING format this is a placeholder like the #.

. The position of the decimal point.

, Separator.

- Stores minus if the number is negative.

+ Stores the sign of the number.

Acknowledgement: the function `format` was implemented by Paulo Soares <psoares@consiste.pt>

25.71 FORMATDATE

```
FormatDate("format",time)
```

Formats a time value (or date) according to the format string. The format string may contain placeholders. The first argument is the format string the second argument is the time value to convert. If the second argument is missing or it is `undef` then the local time is converted.

If the time value is presented it has to be the number of seconds elapsed since January 1, 1970. This is the usual time stamp value generally used under UNIX.

If the second argument is a negativevalue then this is treated relative to the current time point. For example

```
print FormatDate("YEAR MON DD HH:mm:ss", -60)
```

will print out the time that was one minute ago.

25.71.1 FORMATDATE Details

This function uses the first argument as a formatting string. This string may contain the following character strings:

```
YEAR          four digit year
YY            two digit year
MON           three letter abbreviation of the month name
MM            month
0M            month with leading zero if needed
*MONTH-NAME*  name of the month
DD            day of the month
0D            day of the month with leading zero if needed
WD            week day on a single digit starting with sunday=0
WEEKDAY-NAME  the name of the weekday
WDN           three letter abbreviation fo the week day name
HH            hours (24 hours notation)
0H            hours with leading zero if needed (24 hours notation)
hh            hours (12 hours notation)
```

```
0h          hours with leading zero if needed (12 hours notation)
mm          minutes
0m          minutes with leading zero if needed
ss          seconds
0s          seconds with leading zeroes if needed
am
pm   is am or pm
```

Any other character in the format string will get into the result verbatim.

25.72 FILEOWNER(FileName)

This function returns the name of the owner of a file as a string. If the file does not exist or for some other reason the owner of the file can not be determined then the function returns `undef`.

This function returns the name of the owner of a file as a string. If the file does not exist or for some other reason the owner of the file can not be determined then the function returns `undef`.

This function returns the name of the owner of a file as a string. If the file does not exist or for some other reason the owner of the file can not be determined then the function returns `undef`.

25.73 FRAC

The function returns the fractional part of the argument. This function always returns a double except that `FRAC(undef)` may return `undef`. `FRAC(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

Negative arguments return negative value (or zero if the argument is a negative integer), positive arguments result positive values (or zero if the argument is integer).

25.74 FREEFILE()

This function returns a free file number, which is currently not associated with any opened file. If there is no such file number it returns `undef`.

The returned value can be used in a consecutive See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement to specify a file number. Another way to get a free file number is to set a variable to hold the integer value zero and use the variable as file number in the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. For more information on this method see the documentation of the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. This function returns a free file number, which is currently not associated with any opened file. If there is no such file number it returns `undef`.

The returned value can be used in a consecutive See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement to specify a file number. Another way to get a free file number is to set a variable to hold the integer value zero and use the variable as file number in the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. For more information on this method see the

documentation of the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. This function returns a free file number, which is currently not associated with any opened file. If there is no such file number it returns `undef`.

The returned value can be used in a consecutive See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement to specify a file number. Another way to get a free file number is to set a variable to hold the integer value zero and use the variable as file number in the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. For more information on this method see the documentation of the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩.

25.75 FUNCTION fun()

This command should be used to define a function. A function is a piece of code that can be called by the BASIC program from the main part or from a function or subroutine.

```
FUNCTION fun(a,b,c)
...
fun = returnvalue
...
END FUNCTION
```

The end of the function is defined by the line containing the keywords `END FUNCTION`.

25.75.1 FUNCTION Details

The function declaration can be placed anywhere in the code before or after the function is used. Because the functions just as well as the variables are typeless there is no such thing as function prototype and thus functions can not be declared beforehand.

You can call a function in your BASIC code that is not defined yet. The syntax analyzer seeing the `function_name()` construct will see that this is a function call and will generate the appropriate code to call the function that is defined later. If the function is not defined until the end of the code then the interpreter will generate error before the code is started.

Functions just as well as subroutines can have argument and local variables. When the function or subroutine is called the actual values are evaluated and placed in the argument variables. It is not an error when calling a function to specify less or more number of actual arguments than the declared arguments.

If the actual arguments are too few the rest of the arguments will become `undef` before the function starts. If there are too many arguments when calling a function or subroutine the extra arguments are evaluated and then dropped (ignored).

ScriptBasic allows recursive functions thus a function can call itself, but a function or subroutine is not allowed to be declared inside of another subroutine or function.

Functions differ from subroutines in the fact functions return value while subroutines do not. However in ScriptBasic this is not a strict rule. Subroutines declared with `SUB` instead of `FUNCTION` are allowed to return a value and been used just as if it was declared with the keyword `FUNCTION`.

To return a value from a function the name of the function can be used just like a local variable. The assignment assigning a value to the function will be returned from

the function. If there are more than one assignments to the function name then the last assignment executed will prevail. On the other hand you can not use the function name as a local variable and retrieve the last assigned value from the function name. Using the function name inside the function in an expression will result recursive call or will be used as a global or local variable.

If the actual value of an argument is left value then the reference to the actual argument is passed to the function or the subroutine instead of the value. In other cases the value of the expression standing in the position of the argument when calling the function is passed.

Passing a variable by reference means altering the argument variable also alters the variable that was passed to the function.

To force a variable passed by value rather than reference the operator `ByVal` can be used. When `ByVal` is used as a numeric operator in an expression acts as an identity operator. It means that the value of the expression is the same as the value standing on the right of the operator `ByVal`. However this is already an expression and not a variable and as such it can not be passed by reference only the value.

The keyword `ByVal` can also be used as a command listing all the argument variables after the keyword on a line:

```
function myfunc(a,b,c,d)
ByVal a,b,c
...
```

In this case `ByVal` acts as a command and replaces the references by copies of the values. After executing this command the argument values can be altered without affecting the variables that stand in the argument's place where the function is called.

Although the command `ByVal` looks like a declaration of arguments to be passed by value instead of reference; this is not a declaration but it rather a command that has to be executed.

25.76 GCD

This is a planned function that takes two or more integer argument and calculates the largest common divisor of them.

25.77 GMTIME

This function returns the GMT time expressed as seconds since January 1, 1970, 00:00am. The function does not accept any argument. This function is similar to the function See ⟨undefined⟩ [NOW], page ⟨undefined⟩ but returns the GMT time instead of the actual local time.

25.78 GMTOLOCALTIME

This function accepts one argument that has to be the number of seconds elapsed since January 1, 1970 0:00 am in GMT. The function returns the same number of seconds in local time. In other words the function converts a GMT time value to local time value.

25.79 GOSUB label

=H Gosub commands

This is the good old way implementation of the BASIC `GOSUB` command. The command `GOSUB` works similar to the command `GOTO` with the exception that the next return command will drive the interpreter to the line following the line with the `GOSUB`.

You can only call a code segment that is inside the actual code environment. In other words if the `GOSUB` is in a function or subroutine then the label referenced by the `GOSUB` should also be in the same function or subroutine. Similarly any `GOSUB` in the main code should reference a label, which is also in the main code.

To return from the code fragment called by the command `GOSUB` the command `RETURN` should be used. Note that this will not break the execution of a function or a subroutine. The execution will continue on the command line following the `GOSUB` line.

`GOSUB` commands can follow each other, ScriptBasic will build up a stack of `GOSUB` calls and will return to the appropriate command line following the matching `GOSUB` command.

When a subroutine or function contains `GOSUB` commands and the function or subroutine is finished so that one or more executed `GOSUB` command remains without executed `RETURN` then the `GOSUB/RATURN` stack is cleared. This is not an error.

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

25.80 GOTO label

Go to a label and continue program execution at that label. Labels are local within functions and subroutines. You can not jump into a subroutine or jump out of it.

Use of GOTO is usually discouraged and is against structural programming. Whenever you feel the need to use the `GOTO` statement (except `ON ERROR GOTO`) thin it twice whether there is a better solution without utilizing the statement `GOTO`.

Typical use of the `GOTO` statement to jump out of some error condition to the error handling code or jumping out of some loop on some condition.

25.81 HCOS

This is a planned function to calculate the cosinus hyperbolicus of the argument.

25.82 HCOSECANT

This is a planned function to calculate the cosecant hyperbolicus of the argument.

25.83 HCTAN

This is a planned function to calculate the cotangent hyperbolicus of the argument.

25.84 HEX(n)

Take the argument as a long value and convert it to a string that represents the value in hexadecimal form. The hexadecimal form will contain upper case alpha character if there is any alpha character in the hexadecimal representation of the number.

25.85 HOSTNAME()

This function accepts no argument and returns the host name of the machine executing the BASIC program.

This function accepts no argument and returns the host name of the machine executing the BASIC program. This host name is the TCP/IP network host name of the machine.

This function accepts no argument and returns the host name of the machine executing the BASIC program.

25.86 HOUR

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the hour value of that time. If the argument is missing the function uses the actual local time.

25.87 HSECANT

This is a planned function to calculate the secant hyperbolicus of the argument.

25.88 HSIN

This is a planned function to calculate the sinus hyperbolicus of the argument.

25.89 HTAN

This is a planned function to calculate the tangent hyperbolicus of the argument.

25.90 ICALL n,v1,v2, ... ,vn

ICALL is implicit call. The first argument of an `ICALL` command or `ICALL` function should be the integer value returned by the function See ⟨undefined⟩ [ADDRESS], page ⟨undefined⟩ as the address of a user defined function.

The rest of the arguments are the arguments to be passed to the function to be called. The return value if the function `ICALL` is the value of the implicitly called function.

25.90.1 ICALL Details

Whenever you call a function or subroutine you have to know the name of the subroutine or function. In some situation programmers want to call a function without knowing the name of the function. For example you want to write a sorting subroutine that sorts general elements and the caller should provide a subroutine that makes the comparison. This way the sorting algorithm can be implemented only once and need not be rewritten each time a new type of data is to be sorted. The sorting subroutine gets the comparing function as an argument and calls the function indirectly. ScriptBasic can not pass functions as arguments to other functions, but it can pass integer numbers. The function See ⟨undefined⟩ [ADDRESS], page ⟨undefined⟩ can be used to convert a function into integer. The result of the built-in function See ⟨undefined⟩ [ADDRESS], page ⟨undefined⟩ is an integer number, which is associated inside the basic code with the function. You can pass this value to the `ICALL` command or function as first argument. The `ICALL` command is the command for indirect subroutine call. The call

```
ICALL Address(MySubroutine()),arg1,arg2,arg3
```

is equivalent to

```
CALL MySubroutine( arg1,arg2,arg3)
```

If you call a function that has return value use can use the `ICALL` function instead of the `ICALL` statement:

```
A = ICALL(Address(MyFunction()),arg1,arg2,arg3)
```

is equivalent to

```
A = MyFunction(arg1,arg2,arg3)
```

The real usage of the function Address and icall can be seen in the following example:

```
sub MySort(sfun,q)
local ThereWasNoChange,SwapVar
repeat
 ThereWasNoChange = 1
 for i=lbound(q) to ubound(q)-1

  if  icall(sfun,q[i],q[i+1]) > 0 then
   ThereWasNoChange = 0
   SwapVar = q[i]
   q[i] = q[i+1]
   q[i+1] = SwapVar
  endif

 next i
until ThereWasNoChange

end sub

function IntegerCompare(a,b)
  if a < b then
   cmp = -1
```

```
  elseif a = b then
   cmp = 0
  else
   cmp = 1
  endif
end function

h[0] = 2
h[1] = 7
h[2] = 1

MySort address(IntegerCompare()) , h

for i=lbound(h) to ubound(h)
 print h[i],"\n"
next i
```

Note that the argument of the function Address is a function call. ScriptBasic allows variables and functions to share the same name. Address is a built-in function just as any other built in function, and therefore the expression

```
Address(MySub) B<THIS IS WRONG!>
```

Is syntactically correct. The only problem is that it tries to calculate the address of the variable `MySub`, which it can not and results a run-time error. Instead you have to write

```
Address( MySub() )
```

using the parentheses. In this situation the function or subroutine `MySub()` will not be invoked. The parentheses tell the compiler that this is a function and not a variable.

25.91 IF condition THEN

Conditional execution. There are two different ways to use this command: single line `IF` and multi line `IF`.

A single line `IF` has the form

```
IF condition THEN command
```

There is no way to specify any `ELSE` part for the command in the single line version. If you need `ELSE` command you have use multi line `IF`.

The multi line `IF` should not contain any command directly after the keyword `THEN`. It should have the format:

```
IF condition THEN
  commands
ELSE
  commands
END IF
```

The `ELSE` part of the command is optional, thus the command can have the format

```
IF condition THEN
  commands
```

```
END IF
```

as well. To be very precise the full syntax of the multi-line `IF` command is:

```
IF condition THEN
 commands
[ ELSE IF | ELSEIF | ELSIF | ELIF
  commands
  ... ]
[ ELSE
  commands ]
END IF | ENDIF
```

You can use as many `ELSE IF` branches as you like and at most one `ELSE` branch.

The keywords `ELSE IF`, `ELSEIF` and others are allowed for ease program porting from other BASIC dialect. There is no difference between the interpretation. The same is true for `END IF` in two words and written into a single keyword `ENDIF`.

25.92 IMAX

This is a planned function to select and return the index of the maximum of the arguments.

25.93 IMIN

This is a planned function to select and return the index of the minimum of the arguments.

25.94 INPUT(n,fn)

This function reads records from an opened file.

Arguments:

`n` the first argument is the number of records to read. The size of the record in terms of bytes is defined as the `LEN` parameter when the file was opened. If this was missing the function reads `n` bytes from the file or socket.

`fn` the second parameter is the file number associated to the opened file by the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. If this parameter is missing the function reads from the standard input.

The function tries but not necessarily reads `n` records from the file. To get the actual number of bytes (and not records!) read from the file you can use the function `LEN` on the result string.

Note that some Basic languages allow the form

```
a = INPUT(20,#1)
```

however this extra `#` is not allowed in ScriptBasic. The character `#` is an operator in ScriptBasic defined as no-operation and therefore you can use this form. On the other

hand operators like `#` are reserved for the external modules and some external module may redefine this operator. Programs using such modules may end up in trouble when using the format above therefore it is recommended not to use the above format.

This function reads records from an opened file.

Arguments:

`n` the first argument is the number of records to read. The size of the record in terms of bytes is defined as the `LEN` parameter when the file was opened. If this was missing the function reads `n` bytes from the file or socket.

`fn` the second parameter is the file number associated to the opened file by the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. If this parameter is missing the function reads from the standard input.

The function tries but not necessarily reads `n` records from the file. To get the actual number of bytes (and not records!) read from the file you can use the function `LEN` on the result string.

Note that some Basic languages allow the form

```
a = INPUT(20,#1)
```

however this extra `#` is not allowed in ScriptBasic. The character `#` is an operator in ScriptBasic defined as no-operation and therefore you can use this form. On the other hand operators like `#` are reserved for the external modules and some external module may redefine this operator. Programs using such modules may end up in trouble when using the format above therefore it is recommended not to use the above format.

This function reads records from an opened file.

Arguments:

`n` the first argument is the number of records to read. The size of the record in terms of bytes is defined as the `LEN` parameter when the file was opened. If this was missing the function reads `n` bytes from the file or socket.

`fn` the second parameter is the file number associated to the opened file by the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. If this parameter is missing the function reads from the standard input.

The function tries but not necessarily reads `n` records from the file. To get the actual number of bytes (and not records!) read from the file you can use the function `LEN` on the result string.

Note that some Basic languages allow the form

```
a = INPUT(20,#1)
```

however this extra `#` is not allowed in ScriptBasic. The character `#` is an operator in ScriptBasic defined as no-operation and therefore you can use this form. On the other hand operators like `#` are reserved for the external modules and some external module may redefine this operator. Programs using such modules may end up in trouble when using the format above therefore it is recommended not to use the above format.

25.95 INSTR(base_string,search_string [,position])

This function can be used to search a sub-string in a string. The first argument is the string we are searching in. The second argument is the string that we actually want to find in the first argument. The third optional argument is the position where the search is to be started. If this argu-ment is missing the search starts with the first character position of the string. The function returns the position where the sub-string can be found in the first string. If the searched sub-string is not found in the string then the return value is undef.

See See ⟨undefined⟩ [INSTRREV], page ⟨undefined⟩

25.96 INSTRREV(base_string,search_string [,position])

This function can be used to search a sub-string in a string in reverse order starting from the end of the string. The first argument is the string we are searching in. The second argument is the string that we actually want to find in the first argument. The third optional argument is the position where the search is to be started. If this argument is missing the search starts with the last character position of the string. The function returns the position where the sub-string can be found in the first string. If the searched sub-string is not found in the string then the return value is undef.

See See ⟨undefined⟩ [INSTR], page ⟨undefined⟩

25.97 INT

This function returns the integral part of the argument. `INT(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`. Other than this the function returns integer value.

The difference between `INT` and `FIX` is that `INT` truncates down while `FIX` truncates towards zero. The two functions are identical for positive numbers. In case of negative arguments `INT` will give a smaller number if the argument is not integer. For example:

```
int(-3.3) = -4
fix(-3.3) = -3
```

See See ⟨undefined⟩ [FIX], page ⟨undefined⟩.

25.98 ISARRAY

This function can be used to determine whether a variable holds array value or ordinary value. If the variable passed as argument to the function is an array then the function returns `true`, otherwise the function returns `false`.

See also See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.99 ISDEFINED

This function can be used to determine whether an expression is defined or undefined (aka `undef`). If the argument is a defined value then the function returns `true`, otherwise the function returns `false`.

This function is the counter function of See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.100 ISDIRECTORY(file_name)

Returns `true` if the named file is a directory and `false` if the file is NOT a directory. Returns `true` if the named file is a directory and `false` if the file is NOT a directory. Returns `true` if the named file is a directory and `false` if the file is NOT a directory.

25.101 ISEMPTY

This function can be used to determine whether an expression holds an empty string. Because programmers tend to use the value `undef` where empty string would be more precise the function returns `true` if the argument is `undef`. Precisely:

The function returns true if the argument is `undef` or a string containing zero characters. Otherwise the function returns `false`.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.102 ISINTEGER

This function can be used to determine whether an expression is integer or some other type of value. If the argument is an integer then the function returns `true`, otherwise the function returns `false`.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.103 ISNUMERIC

This function can be used to determine whether an expression is numeric (real or integer) or some other type of value. If the argument is a real or an integer then the function returns `true`, otherwise the function returns `false`.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.104 ISREAL

This function can be used to determine whether an expression is real or some other type of value. If the argument is a real then the function returns `true`, otherwise the function returns `false`.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.105 ISFILE(file_name)

Returns `true` if the named file is a regular file and `false` if it is a directory. Returns `true` if the named file is a regular file and `false` if it is a directory. Returns `true` if the named file is a regular file and `false` if it is a directory.

25.106 ISSTRING

This function can be used to determine whether an expression is string or some other type of value. If the argument is a string then the function returns `true`, otherwise the function returns `false`.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.107 ISUNDEF

This function can be used to determine whether an expression is defined or undefined (aka `undef`). If the argument is a defined value then the function returns `false`, otherwise the function returns `true`.

This function is the counter function of See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩, See ⟨undefined⟩ [TYPE], page ⟨undefined⟩.

25.108 JOIN(joiner,str1,str2,...)

Join the argument strings using the first argument as a joiner string.

25.108.1 JOIN Details

This function can be used to join several strings together. The first argument of the function is the string used to join the rest of the arguments. The rest of the argument are joined together, but also elements on an array can be joined together. See the example:

```
for i=1 to 8
 q[i] = I
next
print join("|",q)
print
print join("/",1,2,3,4,5,6,7,8)
print
print join(" j-s ",q,2,3,4,5,6,7,8)
print
print join("/",1)
print
```

will print

```
1|2|3|4|5|6|7|8
1/2/3/4/5/6/7/8
1 j-s 2 j-s 3 j-s 4 j-s 5 j-s 6 j-s 7 j-s 8
1
```

The first join joins the elements of the array. The second join joins the arguments of the function. The third example also joins the arguments although the second argument is an array. Because there are more arguments each of them is treated as single value and are joined. Whenever an array is used in place of a single value, the first element of the array is taken. In this example this is 1. The last join is a special one. In this case the join string is not used, because there is only one argument after the join string. Because this argument is not an array there are no elements of it to join.

25.109 JOKER(n)

Return the actual match for the n-th joker character from the last executed See ⟨undefined⟩ [LIKE], page ⟨undefined⟩ operator.

25.109.1 JOKER Details

When a `LIKE` operator is executed ScriptBasic stores the actual strings that matched the joker and wild card characters from the pattern in an array. Using this function the programcan access the actual value of the n-th string that was matched against the n-th joker or wild card character. For example:

```
Const nl="\n"
if "file.txt" like "*.*" then
  print "File=",joker(1)," extension=",joker(2),nl
else
  print "did not match"
endif
```

will print

```
File=file extension=txt
```

25.110 KILL(pid)

This function kills (terminates) a process given by the `pid` and returns true if the process was successfully killed. Otherwise it returns false.

Programs usually want to kill other processes that were started by themselves (by the program I mean) and do not stop. For example you can start an external program using the BASIC command See ⟨undefined⟩ [EXECUTE], page ⟨undefined⟩ to run up to a certain time. If the program does not finish its work and does not stop during this time then that program that started it can assume that the external program failed and got into an infinite loop. To stop this external program the BASIC program should use the function `KILL`.

The BASIC program however can try to kill just any process that runs on the system not only those that were started by the program. It can be successful if the program has the certain permissions to kill the given process.

You can use this function along with the functions See ⟨undefined⟩ [SYSTEM], page ⟨undefined⟩ and `EXECUTE`. You can list the processes currently running on an NT box using some of the functions of the module `NT`.

This function kills (terminates) a process given by the `pid` and returns true if the process was successfully killed. Otherwise it returns false.

Programs usually want to kill other processes that were started by themselves (by the program I mean) and do not stop. For example you can start an external program using the BASIC command See ⟨undefined⟩ [EXECUTE], page ⟨undefined⟩ to run up to a certain time. If the program does not finish its work and does not stop during this time then that program that started it can assume that the external program failed and got into an infinite loop. To stop this external program the BASIC program should use the function `KILL`.

The BASIC program however can try to kill just any process that runs on the system not only those that were started by the program. It can be successful if the program has the certain permissions to kill the given process.

You can use this function along with the functions See ⟨undefined⟩ [SYSTEM], page ⟨undefined⟩ and `EXECUTE`. You can list the processes currently running on an NT box using some of the functions of the module `NT`.

This function kills (terminates) a process given by the `pid` and returns true if the process was successfully killed. Otherwise it returns false.

Programs usually want to kill other processes that were started by themselves (by the program I mean) and do not stop. For example you can start an external program using the BASIC command See ⟨undefined⟩ [EXECUTE], page ⟨undefined⟩ to run up to a certain time. If the program does not finish its work and does not stop during this time then that program that started it can assume that the external program failed and got into an infinite loop. To stop this external program the BASIC program should use the function `KILL`.

The BASIC program however can try to kill just any process that runs on the system not only those that were started by the program. It can be successful if the program has the certain permissions to kill the given process.

You can use this function along with the functions See ⟨undefined⟩ [SYSTEM], page ⟨undefined⟩ and `EXECUTE`. You can list the processes currently running on an NT box using some of the functions of the module `NT`.

25.111 LBOUND

This function can be used to determine the lowest occupied index of an array. Note that arrays are increased in addressable indices automatically, thus it is not an error to use a lower index that the value returned by the function `LBOUND`. On the other hand all the element having index lower than the returned value are `undef`.

The argument of this function has to be an array. If the argument is an ordinary value, or a variable that is not an array the value returned by the function will be `undef`.

`LBOUND(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

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

25.112 LCASE()

Lowercase a string.

25.113 LCM

This is a planned function that takes two or more integer argument and calculates the least common multiple of them.

25.114 LEFT(string,len)

Creates the left of a string. The first argument is the string. The second argument is the number of characters that should be put into the result. If this value is larger than the number of characters in the string then all the string is returned.

See also See ⟨undefined⟩ [MID], page ⟨undefined⟩, See ⟨undefined⟩ [RIGHT], page ⟨undefined⟩

25.114.1 LEFT Details

`left(x,y)` cuts out a substring of `y` characters from the left of the string `x`. If the first argument is not defined the result is also `undef`. Otherwise the first argument is converted to string and the second ar-gument is converted to integer value.

If the second argument is not defined or has negative value it is treated as numeric zero and as such the result string will be empty string.

For compatibility reasons you can append a dollar (`$`) sign to the end of the function identifier.

Example

```
a$ = _
"superqualifragilisticexpialidosys"
print "*",left(a$,undef),"*"
print "*",left(a$,7),"*"
print "*",left(a$,-6),"*"
print "*",left(a$,0),"*"
print left(undef,"66")
```

will print

```
**
*superqu*
**
**
undef
```

25.115 LEN()

This function interprets its argument as a string and returns the length of the string. In ScriptBasic strings can hold any value thus the length of the string is the number of characters contained in the string containing any binary characters, even binary zero.

If the argument is not a string it is converted to string automatically and the length of the converted string is returned. The only exception is `undef` for which the result is also `undef`.

25.116 v = expression

Assign a value to a variable.

On the left side of the `=` a variable or some other ScriptBasic left value has to stand. On the right side an expression should be used. First the left value is evaluated and then the expression. Finally the left value's old value is replaced by the result of the expression.

The left value standing on the left side of the `=` can be a local or global variable, array element or associative array element.

25.117 v &= expression

Append a string to a variable.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v & expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

25.118 v /= expression

Divide a variable by an expression.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v / expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

25.119 v \= expression

Integer divide a variable by a value.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v \ expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

25.120 v -= expression

This command subtracts a value from a variable.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v - expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

25.121 v += expression

Add a value to a variable.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v + expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

25.122 v *= expression

Multiply a variable with a value.

The variable can be a global or local variable, array element or associative array element.

You can use this command as a shorthand for `v = v * expression`. Using this short format is more readable in some cases and generates more efficient code. However note that this kind of assignment operation is a C language like operator and is not common in BASIC programs.

25.123 string LIKE pattern

Compare a string against a pattern.

```
string LIKE pattern
```

The pattern may contain joker characters and wild card characters.

25.123.1 LIKE Details

Pattern matching in ScriptBasic is similar to the pattern matching that you get used to on the UNIX or Windows NT command line. The operator like compares a string to a pattern.

```
string like pattern
```

Both string and pattern are expressions that should evaluate to a string. If the pattern matches the string the result of the operator is true, otherwise the result is false.

The pattern may contain normal characters, wild card characters and joker characters. The normal characters match themselves. The wild card characters match one or more characters from the set they are for. The joker characters match one character from the set they stand for. For example:

```
Const nl="\n"
print "file.txt" like "*.txt",nl
print "file0.txt" like "*?.txt",nl
print "file.text" like "*.txt",nl
```

will print

```
-1
-1
0
```

The wild card character `*` matches a list of characters of any code. The joker character `?` matches a single character of any code. In the first print statement the `*` character matches the string file and `.txt` matches itself at the end of the string. In the second example `*` matches the string file and the joker `?` matches the character `0`. The wild card character `*` is the most general wild card character because it matches one or more of any character. There are other wild card characters. The character `#` matches one or more digits, `$` matches one or more alphanumeric characters and finally `@` matches one or more alpha characters (letters).

```
* all characters
# 0123456789
$ 0123456789abcdefghijklmnopqrstxyvwzABCDEFGHIJKLMNOPQRSTXYVWZ
 abcdefghijklmnopqrstxyvwzABCDEFGHIJKLMNOPQRSTXYVWZ
```

A space in the pattern matches one or more white spaces, but the space is not a regular wild card character, because it behaves a bit different.

Note that wild card character match ONE or more characters and not zero or more as in other systems. Joker characters match exactly one character, and there is only one joker character by default, the character `?`, which matches a single character of any code.

We can match a string to a pattern, but that is little use, unless we can tell what substring the joker or wildcard characters matched. For the purpose the function joker is available. The argument of this function is an integer number, n starting from 1 and the result is the substring that the last pattern matching operator found to match the nth joker or wild card character. For example

```
Const nl="\n"
if "file.txt" like "*.*" then
  print "File=",joker(1)," extension=",joker(2),nl
else
  print "did not match"
endif
```

will print

```
File=file extension=txt
```

If the pattern did not match the string or the argument of the function joker is zero or negative, or is larger than the serial number of the last joker or wild card character the result is `undef`.

Note that there is no separate function for the wild card character substrings and one for the joker characters. The function joker serves all of them counting each from left to right. The function joker does not count, nor return the spaces, because programs usually are not interested in the number of the spaces that separate the lexical elements matched by the pattern.

Sometimes you want a wild card character or joker character to match only itself. For example you want to match the string `"13*52"` to the pattern two numbers separated by a star. The problem is that the star character is a wild card character and therefore `"#*#"` matches any string that starts and ends with a digit. But that may not be a problem. A `*` character matches one or more characters, and therefore `"#*#"` will indeed match `"13*52"`. The problem is, when we want to use the substrings.

```
Const nl="\n"
a="13*52" like "#*#"
print joker(1)," ",joker(3),nl
a="13*52" like "#~*#"
print joker(1)," ",joker(2),nl
```

will print

```
1 52
13 52
```

The first `#` character matches one character, the `*` character matches the substring `"3*"` and the final `#` matches the number `52`.

The solution is the pattern escape character. The pattern escape character is the tilde character: `~`. Any character following the `~` character is treated as normal character and is matched only by itself. This is true for any normal character, for wild card characters; joker characters; for the space and finally for the tilde character itself. The space character following the tilde character matches exactly one space characters.

Pattern matching is not always as simple as it seems to be from the previous examples. The pattern `"*.*"` matches files having extension and `joker(1)` and `joker(2)` can be used to retrieve the file name and the extension. What about the file `sciba_source.tar.gz`? Will it result

```
File=scriba_source.tar extension=gz
```

or

```
File=scriba_source extension=tar.gz
```

The correct result is the second. Wild card characters implemented in ScriptBasic are not greedy. They eat up only as many characters as they need.

Up to now we were talking about wild card characters and the joker character defining what matches what as final rule carved into stone. But these rules are only the default behavior of these characters and the program can alter the set of characters that a joker or wild card character matches.

There are 13 characters that can play joker or wild card character role in pattern matching. These are:

```
*  #  $  ?  &  %  !  +  /  |  <  >
```

When the program starts only the first five characters have special meaning the others are normal characters. To change the role of a character the program has to execute a set joker or set wild command. The syntax of the commands are:

set joker expression to expression set wild expression to expression

Both expressions should evaluate to string. The first character of the first string should be the joker or wild card character and the second string should contain all the characters that the joker or wild card character matches. The command set joker alters the behavior of the character to be a joker character matching a single character in the compared string. The command set wild alters the behavior of the character to be a wild card character matching one or more characters in the compared string. For ex-ample if you may want the & character to match all hexadecimal characters the program has to execute:

set wild "&" to "0123456789abcdefABCDEF"

If a character is currently a joker of wild card character you can alter it to be a normal character issuing one of the commands

set no joker expression set no wild expression

where expression should evaluate to a string and the first character of the string should give the character to alter the behavior of.

The two commands are identical, you may always use one or the other; you can use set no joker for a character being currently wild card character and vice versa. You can execute

the command even if the character is currently a normal character in the pattern matching game.

Using the commands now we can see that

25.124 LINE INPUT

Read a line from a file or from the standard input.

The syntax of the command is

```
LINE INPUT [# i , ] variable
```

The parameter `i` is the file number used in the open statement. If this is not specified the standard input is read.

The `variable` will hold a single line from the file read containing the possible new line character terminating the line. If the last line of a file is not terminated by a new line character then the `variable` will not contain any new line character. Thus this command does return only the characters that are really in the file and does not append extra new line character at the end of the last line if that lacks it.

On the other hand you should not rely on the missing new line character from the end of the last line because it may and usually it happens to be there. Use rather the function See ⟨undefined⟩ [EOF], page ⟨undefined⟩ to determine if a file reading has reached the end of the file or not.

See also See ⟨undefined⟩ [CHOMP], page ⟨undefined⟩

You can also read from sockets using this command but you should be careful because data in a socket comes from programs generated on the fly. This means that the socket pipe may not contain the line terminating new line and not finished as well unlike a file. Therefore the command may start infinitely long when trying to read from a socket until the application on the other end of the line sends a new line character or closes the socket. When you read from a file this may not happen.

Read a line from a file or from the standard input.

The syntax of the command is

```
LINE INPUT [# i , ] variable
```

The parameter `i` is the file number used in the open statement. If this is not specified the standard input is read.

The `variable` will hold a single line from the file read containing the possible new line character terminating the line. If the last line of a file is not terminated by a new line character then the `variable` will not contain any new line character. Thus this command does return only the characters that are really in the file and does not append extra new line character at the end of the last line if that lacks it.

On the other hand you should not rely on the missing new line character from the end of the last line because it may and usually it happens to be there. Use rather the function See ⟨undefined⟩ [EOF], page ⟨undefined⟩ to determine if a file reading has reached the end of the file or not.

See also See ⟨undefined⟩ [CHOMP], page ⟨undefined⟩

You can also read from sockets using this command but you should be careful because data in a socket comes from programs generated on the fly. This means that the socket pipe may not contain the line terminating new line and not finished as well unlike a file. Therefore the command may start infinitely long when trying to read from a socket until the application on the other end of the line sends a new line character or closes the socket. When you read from a file this may not happen.

Read a line from a file or from the standard input.

The syntax of the command is

```
LINE INPUT [# i , ] variable
```

The parameter `i` is the file number used in the open statement. If this is not specified the standard input is read.

The `variable` will hold a single line from the file read containing the possible new line character terminating the line. If the last line of a file is not terminated by a new line character then the `variable` will not contain any new line character. Thus this command does return only the characters that are really in the file and does not append extra new line character at the end of the last line if that lacks it.

On the other hand you should not rely on the missing new line character from the end of the last line because it may and usually it happens to be there. Use rather the function See ⟨undefined⟩ [EOF], page ⟨undefined⟩ to determine if a file reading has reached the end of the file or not.

See also See ⟨undefined⟩ [CHOMP], page ⟨undefined⟩

You can also read from sockets using this command but you should be careful because data in a socket comes from programs generated on the fly. This means that the socket pipe may not contain the line terminating new line and not finished as well unlike a file. Therefore the command may start infinitely long when trying to read from a socket until the application on the other end of the line sends a new line character or closes the socket. When you read from a file this may not happen.

25.125 LOC()

Return current file pointer position of the opened file. The argument of the function is the file number that was used by the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ opening the file.

This function is the counter part of the statement See ⟨undefined⟩ [SEEK], page ⟨undefined⟩ that sets the file pointer position.

The file position is counted in record size. This means that the file pointer stands after the record returned by the function. This is not necessarily stands right after the record at the start of the next record actually. It may happen that the file pointer stands somewhere in the middle of the next record. Therefore the command

```
SEEK fn,LOC(fn)
```

may alter the actual file position and can be used to set the file pointer to a safe record boundary position.

If there was no record size defined when the file was opened the location is counted in bytes. In this case the returned value precisely defines where the file pointer is. Return

current file pointer position of the opened file. The argument of the function is the file number that was used by the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ opening the file.

This function is the counter part of the statement See ⟨undefined⟩ [SEEK], page ⟨undefined⟩ that sets the file pointer position.

The file position is counted in record size. This means that the file pointer stands after the record returned by the function. This is not necessarily stands right after the record at the start of the next record actually. It may happen that the file pointer stands somewhere in the middle of the next record. Therefore the command

```
SEEK fn,LOC(fn)
```

may alter the actual file position and can be used to set the file pointer to a safe record boundary position.

If there was no record size defined when the file was opened the location is counted in bytes. In this case the returned value precisely defines where the file pointer is. Return current file pointer position of the opened file. The argument of the function is the file number that was used by the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ opening the file.

This function is the counter part of the statement See ⟨undefined⟩ [SEEK], page ⟨undefined⟩ that sets the file pointer position.

The file position is counted in record size. This means that the file pointer stands after the record returned by the function. This is not necessarily stands right after the record at the start of the next record actually. It may happen that the file pointer stands somewhere in the middle of the next record. Therefore the command

```
SEEK fn,LOC(fn)
```

may alter the actual file position and can be used to set the file pointer to a safe record boundary position.

If there was no record size defined when the file was opened the location is counted in bytes. In this case the returned value precisely defines where the file pointer is.

25.126 LOCATLTOGMTIME

This function accepts one argument that has to be the number of seconds elapsed since January 1, 1970 0:00 am in local time. The function returns the same number of seconds in GMT. In other words the function converts a local time value to GMT time value.

25.127 LOCK # fn, mode

Lock a file or release a lock on a file. The `mode` parameter can be `read`, `write` or `release`.

When a file is locked to `read` no other program is allowed to write the file. This ensures that the program reading the file gets consistent data from the file. If a program locks a file to read using the lock value `read` other programs may also get the `read` lock, but no program can get the See ⟨undefined⟩ [write], page ⟨undefined⟩ lock. This means that any program trying to write the file and issuing the command `LOCK` with the parameter `write` will stop and wait until all read locks are released.

When a program write locks a file no other program can read the file or write the file.

Note that the different operating systems and therefore ScriptBasic running on different operating systems implement file lock in different ways. UNIX operating systems implement so called advisory locking, while Windows NT implements mandatory lock.

This means that a program under UNIX can write a file while another program has a read or write lock on the file if the other program is not good behaving and does not ask for a write lock. Therefore this command under UNIX does not guarantee that any other program is not accessing the file simultaneously.

Contrary Windows NT does lock the file in a hard way, and this means that no other process can access the file in prohibited way while the file is locked.

This different behavior usually does not make harm, but in some rare cases knowing it may help in debugging some problems. Generally you should not have a headache because of this.

You should use this command to synchronize the BASIC programs running parallel and accessing the same file.

You can also use the command `LOCK REGION` to lock a part of the file while leaving other parts of the file accessible to other programs.

If you heavily use record oriented files and file locks you may consider using some data base module to store the data in database instead of plain files.

25.128 LOCK REGION # fn FROM start TO end FOR mode

Lock a region of a file. The region starts with the record `start` and ends with the record `end` including both end positions. The length of a record in the file is given when the file is opened using the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩.

The mode can be `read`, `write` and `release`. The command works similar as whole file locking, thus it is recommended that you read the differences of the operating systems handling locking in the section of file locking for the command `LOCK`. Lock a region of a file. The region starts with the record `start` and ends with the record `end` including both end positions. The length of a record in the file is given when the file is opened using the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩.

The mode can be `read`, `write` and `release`. The command works similar as whole file locking, thus it is recommended that you read the differences of the operating systems handling locking in the section of file locking for the command `LOCK`. Lock a region of a file. The region starts with the record `start` and ends with the record `end` including both end positions. The length of a record in the file is given when the file is opened using the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩.

The mode can be `read`, `write` and `release`. The command works similar as whole file locking, thus it is recommended that you read the differences of the operating systems handling locking in the section of file locking for the command `LOCK`.

25.129 LOF()

This function returns the length of an opened file in number of records. The argument of the function has to be the file number that was used by the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ to open the file.

The actual number of records is calculated using the record size specified when the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ was used. The returned number is the number of records that fit in the file. If the file is longer containing a fractional record at the end the fractional record is not counted.

If there was no record length specified when the file was opened the length of the file is returned in number of bytes. In this case fractional record has no meaning. This function returns the length of an opened file in number of records. The argument of the function has to be the file number that was used by the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ to open the file.

The actual number of records is calculated using the record size specified when the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ was used. The returned number is the number of records that fit in the file. If the file is longer containing a fractional record at the end the fractional record is not counted.

If there was no record length specified when the file was opened the length of the file is returned in number of bytes. In this case fractional record has no meaning. This function returns the length of an opened file in number of records. The argument of the function has to be the file number that was used by the statement See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ to open the file.

The actual number of records is calculated using the record size specified when the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ was used. The returned number is the number of records that fit in the file. If the file is longer containing a fractional record at the end the fractional record is not counted.

If there was no record length specified when the file was opened the length of the file is returned in number of bytes. In this case fractional record has no meaning.

25.130 LOG

Calculates the natural log of the argument. If the argument is zero or less than zero the result is `undef`.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`LOG(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.131 LOG10

Calculates the log of the argument. If the argument is zero or less than zero the result is `undef`

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`LOG10(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.132 LTRIM()

Remove the space from the left of the string.

25.133 MAX

This is a planned function to select and return the maximum of the arguments.

25.134 MAXINT

This built-in constant is implemented as an argument less function. Returns the maximal number that can be stored as an integer value.

25.135 MID(string,start [,len])

Return a subpart of the string. The first argument is the string, the second argument is the start position. The third argument is the length of the sub-part in terms of characters. If this argument is missing then the subpart lasts to the last character of the argument `string`.

See also See ⟨undefined⟩ [LEFT], page ⟨undefined⟩, See ⟨undefined⟩ [RIGHT], page ⟨undefined⟩.

25.135.1 MID Details

`mid(x,y,[z])` cuts out a sub-string from the string `x`. If the first argument of the function is undefined the result is `undef`. Otherwise the first argument is converted to string and the second and third arguments are converted to numeric value. The third argument is optional.

The second argument specifies the start position of the resulting substring in the original string x; and the last argument specifies the number of characters to take from the original string `x`. If the third argument is missing the substring lasts from the start position to the end of the string. If the second argu-ment is not defined the start of the substring is at the start of the original string. In other words if the second argument is missing it is the same as value 1. If the second argument is zero or negative it will specify the start position counting the characters from the end of the string.

If the staring position `y` points beyond the end of the string the result is empty string. If the length of the substring is larger than the number of characters between the starting position and end of the original string then the result will be the substring between the start position and the end of the original string.

If the length of the substring is negative the characters before the starting position are taken. No more than the available characters can be taken in this case either. In other words if the length is negative and is larger in absolute value than the starting position the resulting sub-string is the character between the position specified by the second argument and the start of the string.

Note that the order of the characters is never changed even if some position or length parameters are negative.

For compatibility reasons you can append a dollar ($) sign to the end of the function identifier.

Example:

```
a$ = "superqualifragilisticexpialidosys"
print mid(a$,undef)
print mid(a$,1,5)
print mid(a$,undef,6)
print mid(a$,6,5)
print mid(a$,"-3")
print "*",mid(a$,0),"*"
print mid(undef,"66")
print mid(a$,6,-3)
print mid(a$,6,3)
print mid(a$,-4,-3)
print mid(a$,-4,3)
```

will print

```
superqualifragilisticexpialidosys
super
superq
quali
sys
**
undef
erq
qua
ido
osy
```

25.136 MIN

This is a planned function to select and return the minimum of the arguments.

25.137 MININT

This built-in constant is implemented as an argument less function. Returns the minimal ("maximal negative") number that can be stored as an integer value.

25.138 MINUTE

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the minute value of that time. If the argument is missing it uses the actual local time.

25.139 MKD

This is a planned function to convert the argument real number to an 8 byte string.

Converts the double-precision number "n" into an 8-byte string so it can later be retrieved from a random-access file as a numeric value.

25.140 MKDIR directory_name

This command creates a new directory. If it is needed then the command attempts to create all directories automatically that are needed to create the final directory. For example if you want to create `public_html/cgi-bin` but the directory `public_html` does not exist then the command

```
MKDIR "public_html/cgi-bin"
```

will first create the directory `public_html` and then `cgi-bin` under that directory.

If the directory can not be created for some reason an error is raised.

This is not an error if the directory does already exist.

You need not call this function when you want to create a file using the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. The command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ automatically creates the needed directory when a file is opened to be written.

The created directory can be erased calling the command See ⟨undefined⟩ [DELETE], page ⟨undefined⟩ or calling the dangerous command See ⟨undefined⟩ [DELTREE], page ⟨undefined⟩.

This command creates a new directory. If it is needed then the command attempts to create all directories automatically that are needed to create the final directory. For example if you want to create `public_html/cgi-bin` but the directory `public_html` does not exist then the command

```
MKDIR "public_html/cgi-bin"
```

will first create the directory `public_html` and then `cgi-bin` under that directory.

If the directory can not be created for some reason an error is raised.

This is not an error if the directory does already exist.

You need not call this function when you want to create a file using the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. The command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ automatically creates the needed directory when a file is opened to be written.

The created directory can be erased calling the command See ⟨undefined⟩ [DELETE], page ⟨undefined⟩ or calling the dangerous command See ⟨undefined⟩ [DELTREE], page ⟨undefined⟩.

This command creates a new directory. If it is needed then the command attempts to create all directories automatically that are needed to create the final directory. For example if you want to create `public_html/cgi-bin` but the directory `public_html` does not exist then the command

```
MKDIR "public_html/cgi-bin"
```

will first create the directory `public_html` and then `cgi-bin` under that directory.

If the directory can not be created for some reason an error is raised.

This is not an error if the directory does already exist.

You need not call this function when you want to create a file using the command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩. The command See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ automatically creates the needed directory when a file is opened to be written.

The created directory can be erased calling the command See ⟨undefined⟩ [DELETE], page ⟨undefined⟩ or calling the dangerous command See ⟨undefined⟩ [DELTREE], page ⟨undefined⟩.

25.141 MKI

This is a planned function to convert the argument integer number to an 2 byte string.

Converts the integer number `"n"` into an 2-byte string so it can later be retrieved from a random-access file as a numeric value.

25.142 MKL

This is a planned function.

Converts the long-integer number `"n"` into an 4-byte string so it can later be retrieved from a random-access file as a numeric value.

25.143 MKS

This is a planned function.

Converts the single-precision number `"n"` into an 4-byte string so it can later be retrieved from a random-access file as a numeric value.

25.144 MONTH

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the month (1 to 12) value of that time. If the argument is missing it uses the actual local time. In other words it returns the actual month in this latter case. The months are numbered so that January is 1 and December is 12.

25.145 NAME filename,filename

Rename a file. The first file is the existing one, the second is the new name of the file. You can not move filed from one disk to another using this command. This command merely renames a single file. Also you can not use wild characters in the source or destination file name.

If you can not rename a file for some reason, you can try to use the command See ⟨undefined⟩ [FileCopy], page ⟨undefined⟩ and then delete the old file. This is successful in some of the cases when `NAME` fails, but it is a slower method.

If the file can not be renamed then the command raises error.

Rename a file. The first file is the existing one, the second is the new name of the file. You can not move filed from one disk to another using this command. This command merely renames a single file. Also you can not use wild characters in the source or destination file name.

If you can not rename a file for some reason, you can try to use the command See ⟨undefined⟩ [FileCopy], page ⟨undefined⟩ and then delete the old file. This is successful in some of the cases when `NAME` fails, but it is a slower method.

If the file can not be renamed then the command raises error.

Rename a file. The first file is the existing one, the second is the new name of the file. You can not move filed from one disk to another using this command. This command merely renames a single file. Also you can not use wild characters in the source or destination file name.

If you can not rename a file for some reason, you can try to use the command See ⟨undefined⟩ [FileCopy], page ⟨undefined⟩ and then delete the old file. This is successful in some of the cases when `NAME` fails, but it is a slower method.

If the file can not be renamed then the command raises error.

25.146 NEXTFILE(dn)

Retrieve the next file name from an opened directory list. If there is no more file names it returns `undef`.

See also See ⟨undefined⟩ [OPENDIR], page ⟨undefined⟩ and See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩.

Retrieve the next file name from an opened directory list. If there is no more file names it returns `undef`.

See also See ⟨undefined⟩ [OPENDIR], page ⟨undefined⟩ and See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩.

Retrieve the next file name from an opened directory list. If there is no more file names it returns `undef`.

See also See ⟨undefined⟩ [OPENDIR], page ⟨undefined⟩ and See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩.

25.147 NOW

This function returns the local time expressed as seconds since January 1, 1970, 00:00am. The function does not accept any argument. This function is similar to the function See ⟨undefined⟩ [GMTIME], page ⟨undefined⟩ but returns the local time instead of the actual GMT.

25.148 OCT(n)

Take the argument as a long value and convert it to a string that represents the value in octal form.

25.149 ODD

Return `true` if the argument is an odd number. `ODD(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

See also See ⟨undefined⟩ [EVEN], page ⟨undefined⟩

25.150 ON ERROR GOTO [label | NULL]

Set the entry point of the error handling routine. If the argument is `NULL` then the error handling is switched off.

25.151 ON ERROR RESUME [label | next]

Setting `ON ERROR RESUME` will try to continue execution on the label or on the next statement when an error occurs without any error handling code.

See also See ⟨undefined⟩ [ONERRORGOTO], page ⟨undefined⟩, See ⟨undefined⟩ [RESUME], page ⟨undefined⟩ and See ⟨undefined⟩ [ERROR], page ⟨undefined⟩.

25.152 OPEN file_name FOR mode AS [#] i [LEN=record_length]

Open or create and open a file. The syntax of the line is

```
OPEN file_name FOR mode AS [ # ] i [ LEN=record_length ]
```

The parameters:

`file_name` if the name of the file to be opened. If the mode allows the file to be written the file is created if it did not existed before. If needed, directory is created for the file.

`mode` is the mode the file is opened. It can be:

> `input` open the file for reading. In this mode the file is opened in read only mode and can not be altered using the file number associated with the open file. Using any function or command that tries to write the file will result in error. In this

mode the file has to exist already to open successfully. If the file to be opened for `input` does not exist the command `OPEN` raises an error.

`output` open the file for writing. If the file existed it's content is deleted first and a freshly opened empty file is ready to accept commands and functions to write into the file. When a file is opened this way no function or command trying to read from the file can be used using the file number associated with the file. The file is opened in ASCII mode but the handling mode can be changed to binary any time.

`append` open a possibly existing file and write after the current content. The same conditions apply as in the mode `output`, thus you can not read the file, only write. The file is opened in ASCII mode but the handling mode can be changed to binary any time.

`random` open the file for reading and writing (textual mode). When you open a file using this mode the file can be written and the content of the existing file can be read. The file pointer can be moved back and forth any time using the command See ⟨undefined⟩ [SEEK], page ⟨undefined⟩ and thus quite complex file handling functions can be implemented. If the file did not exist it is created.

`binary` open the file for reading and writing (binary mode). This mode is the same as `random` with the exception that the file is opened in binary mode.

`socket` open a socket. In this case the file name is NOT a file name, but rather an Internet machine name and a port separated by colon, like `www.digital.com:80` You should not specify any method, like `http://` in front of the machine name, as this command opens a TCP socket to the machine's port and the protocol has to be implemented by the BASIC program.

`#i` is the file number. After the file has been opened this number has to be used in later file handling functions and commands, like See ⟨undefined⟩ [CLOSE], page ⟨undefined⟩ to refer to the file. The `#` character is optional and is allowed for compatibility with other BASIC languages. The number can be between 1 and 512. This number is quite big for most of the applications and provides compatibility with VisualBasic.

`record_length` is optional and specify the length of a record in the file. The default record length is 1 byte. File pointer setting commands usually work on records, thus See ⟨undefined⟩ [SEEK], page ⟨undefined⟩, See ⟨undefined⟩ [TRUNCATE], page ⟨undefined⟩ and other commands and functions accept arguments or return values as number of records. The actual record length is not recorded anywhere thus the BASIC program has to remember the actual length of a record in a file. This is not a BASIC program error to open a file with a different record size than it was created, although this may certainly be a programming error.

If the file number is specified as a variable and the variable value is set to integer zero then the command will automatically find a usable file number and set the variable to hold that value. Using any other expression of value integer zero is an error. Open or create and open a file. The syntax of the line is

```
OPEN file_name FOR mode AS [ # ] i [ LEN=record_length ]
```

The parameters:

`file_name` if the name of the file to be opened. If the `mode` allows the file to be written the file is created if it did not existed before. If needed, directory is created for the file.

`mode` is the mode the file is opened. It can be:

`input` open the file for reading. In this mode the file is opened in read only mode and can not be altered using the file number associated with the open file. Using any function or command that tries to write the file will raise error. In this mode the file has to exist already to open successfully. If the file to be opened for `input` does not exist the command raises error.

`output` open the file for writing. If the file existed it's content is deleted first and a freshly opened empty file is ready to accept commands and functions to write into the file. When a file is opened this way no function or command trying to read from the file can be used using the file number associated with the file. The file is opened in ASCII mode but the handling mode can be changed to binary any time.

`append` open a possibly existing file and write after the current content. The same conditions apply as in the mode `output`, thus you can not read the file, only write. The file is opened in ASCII mode but the handling mode can be changed to binary any time.

`random` open the file for reading and writing (textual mode). When you open a file using this mode the file can be written and the content of the existing file can be read. The file pointer can be moved back and forth any time using the command See ⟨undefined⟩ [SEEK], page ⟨undefined⟩ and thus quite complex file handling functions can be implemented. If the file did not exist it is created.

`binary` open the file for reading and writing (binary mode). This mode is the same as `random` with the exception that the file is opened in binary mode.

`socket` open a socket. In this case the file name is NOT a file name, but rather an Internet machine name and a port separated by colon, like `www.digital.com:80` You should not specify any method, like `http://` in front of the machine name, as this command opens a TCP socket to the machine's port and the protocol has to be implemented by the BASIC program.

`#i` is the file number. After the file has been opened this number has to be used in later file handling functions and commands, like See ⟨undefined⟩ [CLOSE], page ⟨undefined⟩ to refer to the file. The `#` character is optional and is allowed for compatibility with other BASIC languages. The number can be between 1 and 512. This number is quite big for most of the applications and provides compatibility with VisualBasic.

`record_length` is optional and specify the length of a record in the file. The default record length is 1 byte. File pointer setting commands usually work on records, thus See ⟨undefined⟩ [SEEK], page ⟨undefined⟩, See ⟨undefined⟩ [TRUNCATE], page ⟨undefined⟩ and other commands and functions accept arguments or return values as number of records. The actual record length is not recorded anywhere thus the BASIC program has to remember the actual length of a record in a file. This is not a BASIC program error to open a file with a different record size than it was created, although this may certainly be a programming error.

If the file number is specified as a variable and the variable value is set to integer zero then the command will automatically find a usable file number and set the variable to hold that value. Using any other expression of value integer zero is an error. Open or create and open a file. The syntax of the line is

```
OPEN file_name FOR mode AS [ # ] i [ LEN=record_length ]
```

The parameters:

`file_name` if the name of the file to be opened. If the mode allows the file to be written the file is created if it did not existed before. If needed, directory is created for the file.

`mode` is the mode the file is opened. It can be:

`input` open the file for reading. In this mode the file is opened in read only mode and can not be altered using the file number associated with the open file. Using any function or command that tries to write the file will result in error. In this mode the file has to exist already to open successfully. If the file to be opened for `input` does not exist the command `OPEN` raises an error.

`output` open the file for writing. If the file existed it's content is deleted first and a freshly opened empty file is ready to accept commands and functions to write into the file. When a file is opened this way no function or command trying to read from the file can be used using the file number associated with the file. The file is opened in ASCII mode but the handling mode can be changed to binary any time.

`append` open a possibly existing file and write after the current content. The same conditions apply as in the mode `output`, thus you can not read the file, only write. The file is opened in ASCII mode but the handling mode can be changed to binary any time.

`random` open the file for reading and writing (textual mode). When you open a file using this mode the file can be written and the content of the existing file can be read. The file pointer can be moved back and forth any time using the command See ⟨undefined⟩ [SEEK], page ⟨undefined⟩ and thus quite complex file handling functions can be implemented. If the file did not exist it is created.

`binary` open the file for reading and writing (binary mode). This mode is the same as `random` with the exception that the file is opened in binary mode.

`socket` open a socket. In this case the file name is NOT a file name, but rather an Internet machine name and a port separated by colon, like `www.digital.com:80` You should not specify any method, like `http://` in front of the machine name, as this command opens a TCP socket to the machine's port and the protocol has to be implemented by the BASIC program.

`#i` is the file number. After the file has been opened this number has to be used in later file handling functions and commands, like See ⟨undefined⟩ [CLOSE], page ⟨undefined⟩ to refer to the file. The `#` character is optional and is allowed for compatibility with other BASIC languages. The number can be between 1 and 512. This number is quite big for most of the applications and provides compatibility with VisualBasic.

`record_length` is optional and specify the length of a record in the file. The default record length is 1 byte. File pointer setting commands usually work on records, thus See ⟨undefined⟩ [SEEK], page ⟨undefined⟩, See ⟨undefined⟩ [TRUNCATE], page ⟨undefined⟩ and other commands and functions accept arguments or return values as number of records. The actual record length is not recorded anywhere thus the BASIC program has to remember the actual length of a record in a file. This is not a BASIC program error to open a file with a different record size than it was created, although this may certainly be a programming error.

If the file number is specified as a variable and the variable value is set to integer zero then the command will automatically find a usable file number and set the variable to hold that value. Using any other expression of value integer zero is an error.

25.153 OPEN DIRECTORY dir_name PATTERN pattern OPTION option AS dn

Open a directory to retrieve the list of files.

`dir_name` is the name of the directory.

`pattern` is a wild card pattern to filter the file list.

`option` is an integer value that can be composed AND-ing some of the following values

`SbCollectDirectories` Collect the directory names as well as file names into the file list.

`SbCollectDots` Collect the virtual . and .. directory names into the list.

`SbCollectRecursively` Collect the files from the directory and from all the directories below.

`SbCollectFullPath` The list will contain the full path to the file names. This means that the file names returned by the function See ⟨undefined⟩ [NextFile], page ⟨undefined⟩ will contain the directory path specified in the open directory statement and therefore can be used as argument to file handling commands and functions.

`SbCollectFiles` Collect the files. This is the default behavior.

`SbSortBySize` The files will be sorted by file size.

`SbSortByCreateTime` The files will be sorted by creation time.

`SbSortByAccessTime` The files will be sorted by access time.

`SbSortByModifyTime` The files will be sorted by modify time.

`SbSortByName` The files will be sorted by name. The name used for sorting is the bare file name without any path.

`SbSortByPath` The files will be sorted by name including the path. The path is the relative to the directory, which is currently opened. This sorting option is different from the value `sbSortByName` only when the value `sbCollectRecursively` is also used.

`SbSortAscending` Sort the file names in ascending order. This is the default behavior.

`SbSortDescending` Sort the file names in descending order.

`SbSortByNone` Do not sort. Specify this value if you do not need sorting. In this case directory opening can be much faster especially for large directories.

`dn` is the directory number used in later references to the opened directory.

Note that this command can execute for a long time and consuming a lot of memory especially when directory listing is requested recursively. When the command is executed it collects the names of the files in the directory or directories as requested and builds up an

internal list of the file names in the memory. The command See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩ uses the list to retrieve the next file name from the list.

This implies to facts:

The function `NEXTFILE` will not ever return a file name that the file was created after, and did not exist when the command `OPEN DIRECTORY` was executed.

Using See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ after the list of the files is not needed as soon as possible is a good idea.

Using a directory number that was already used and not released calling See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ raises an error.

If the list of the files in the directory can not be collected the command raises error.

See also See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ and See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩.

Open a directory to retrieve the list of files.

`dir_name` is the name of the directory.

`pattern` is a wild card pattern to filter the file list.

`option` is an integer value that can be composed AND-ing some of the following values

`SbCollectDirectories` Collect the directory names as well as file names into the file list.

`SbCollectDots` Collect the virtual . and .. directory names into the list.

`SbCollectRecursively` Collect the files from the directory and from all the directories below.

`SbCollectFullPath` The list will contain the full path to the file names. This means that the file names returned by the function See ⟨undefined⟩ [NextFile], page ⟨undefined⟩ will contain the directory path specified in the open directory statement and therefore can be used as argument to file handling commands and functions.

`SbCollectFiles` Collect the files. This is the default behavior.

`SbSortBySize` The files will be sorted by file size.

`SbSortByCreateTime` The files will be sorted by creation time.

`SbSortByAccessTime` The files will be sorted by access time.

`SbSortByModifyTime` The files will be sorted by modify time.

`SbSortByName` The files will be sorted by name. The name used for sorting is the bare file name without any path.

`SbSortByPath` The files will be sorted by name including the path. The path is the relative to the directory, which is currently opened. This sorting option is different from the value `sbSortByName` only when the value `sbCollectRecursively` is also used.

`SbSortAscending` Sort the file names in ascending order. This is the default behavior.

`SbSortDescending` Sort the file names in descending order.

`SbSortByNone` Do not sort. Specify this value if you do not need sorting. In this case directory opening can be much faster especially for large directories.

`dn` is the directory number used in later references to the opened directory.

Note that this command can execute for a long time and consuming a lot of memory especially when directory listing is requested recursively. When the command is executed it collects the names of the files in the directory or directories as requested and builds up an internal list of the file names in the memory. The command See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩ uses the list to retrieve the next file name from the list.

This implies to facts:

The function `NEXTFILE` will not ever return a file name that the file was created after, and did not exist when the command `OPEN DIRECTORY` was executed.

Using See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ after the list of the files is not needed as soon as possible is a good idea.

Using a directory number that was already used and not released calling See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ raises an error.

If the list of the files in the directory can not be collected the command raises error.

See also See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ and See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩.

Open a directory to retrieve the list of files.

`dir_name` is the name of the directory.

`pattern` is a wild card pattern to filter the file list.

`option` is an integer value that can be composed AND-ing some of the following values

`SbCollectDirectories` Collect the directory names as well as file names into the file list.

`SbCollectDots` Collect the virtual . and .. directory names into the list.

`SbCollectRecursively` Collect the files from the directory and from all the directories below.

`SbCollectFullPath` The list will contain the full path to the file names. This means that the file names returned by the function See ⟨undefined⟩ [NextFile], page ⟨undefined⟩ will contain the directory path specified in the open directory statement and therefore can be used as argument to file handling commands and functions.

`SbCollectFiles` Collect the files. This is the default behavior.

`SbSortBySize` The files will be sorted by file size.

`SbSortByCreateTime` The files will be sorted by creation time.

`SbSortByAccessTime` The files will be sorted by access time.

`SbSortByModifyTime` The files will be sorted by modify time.

`SbSortByName` The files will be sorted by name. The name used for sorting is the bare file name without any path.

`SbSortByPath` The files will be sorted by name including the path. The path is the relative to the directory, which is currently opened. This sorting option is different from the value `sbSortByName` only when the value `sbCollectRecursively` is also used.

`SbSortAscending` Sort the file names in ascending order. This is the default behavior.

`SbSortDescending` Sort the file names in descending order.

`SbSortByNone` Do not sort. Specify this value if you do not need sorting. In this case directory opening can be much faster especially for large directories.

`dn` is the directory number used in later references to the opened directory.

Note that this command can execute for a long time and consuming a lot of memory especially when directory listing is requested recursively. When the command is executed it collects the names of the files in the directory or directories as requested and builds up an internal list of the file names in the memory. The command See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩ uses the list to retrieve the next file name from the list.

This implies to facts:

The function `NEXTFILE` will not ever return a file name that the file was created after, and did not exist when the command `OPEN DIRECTORY` was executed.

Using See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ after the list of the files is not needed as soon as possible is a good idea.

Using a directory number that was already used and not released calling See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ raises an error.

If the list of the files in the directory can not be collected the command raises error.

See also See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩ and See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩.

25.154 OPTION symbol value

Set the integer value of an option. The option can be any string without the double quote. Option names are case insensitive in ScriptBasic.

This command has no other effect than storing the integer value in the option symbol table. The commands or extenal modules may access the values and may change their behavior accoring to the actual values associated with option symbols.

You can retrieve the actual value of an option symbol using the function See ⟨undefined⟩ [OPTIONF], page ⟨undefined⟩

25.155 OPTION("symbol")

Retrieve the actual value of an option symbol as an integer or `undef` if the option was not set. Unlike in the command See ⟨undefined⟩ [OPTION], page ⟨undefined⟩ the argument of this function should be double quoted.

25.156 pack("format",v1,v2,...,vn)

Pack list of arguments into a binary string.

The format strings can contain the packing control literals. Each of these characters optionally take the next argument and convert to the specific binary string format. The result is the concatenated sum of these strings.

Some control characters do not take argument, but result a constant string by their own.

`SZ` the argument is stored as zero terminated string. If the argument already contains zchar that is taken as termnator and the rest of the string is ignored.

`S1` the argument is stored as a string. One byte length and maximum 255 byte strings. If the argument longer than 255 bytes only the first 255 bytes are used, and the rest is ignored.

`S2` same as `S1` but with two bytes for the length.

`S3` same as `S1` but with three bytes for the length.

`S4` same as `S1` but with four bytes for the length.

`S5..8` the same as `S1` but with 5..8 bytes for the length.

`Zn` one or more zero characters, does not take argument. `n` can be 1,2,3 ... positive numbers

`In` integer number stored on n bytes. Low order byte first. If the number does not fit into n bytes the higher bytes are chopped. If the number is negative the high overflow bytes are filled with FF.

`C` character (same as `I1`)

`Un` same as `In` but for unsigned numbers.

`An` store the argument as string on n bytes. If the argument is longer than n bytes only the first n bytes are stored. If the argument is shorter than n bytes the higher bytes are filled with space.

`R` a real number.

See also See ⟨undefined⟩ [UNPACK], page ⟨undefined⟩

25.157 PAUSE

This is a planned command.

```
PAUSE n
```

Suspend the execution of the interpreter (process or thread) for `n` milliseconds.

25.158 PI

This built-in constant is implemented as an argument less function. Returns the approximate value of the constant PI which is the ratio of the circumference of a circle to its diameter.

25.159 POP

Pop off one value from the GOSUB/RETURN stack. After this command a `RETURN` will return to one level higher and to the place where it was called from. For more information see the documentation of the command See ⟨undefined⟩ [GOSUB], page ⟨undefined⟩ and See ⟨undefined⟩ [RETURN], page ⟨undefined⟩.

25.160 POW

Calculates the x-th exponent of 10. If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`POW(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.161 PRINT [# fn ,] print_list

This command prints the elements of the `print_list`. The argument `print_list` is a comma separated list of expressions. The expressions are evaluated one after the other and are printed to the standard output or to the file.

The command prints the `print_list` to an opened file given by the file number `fn`. If `fn` (along with the `#` character) is not specified the command prints to the standard output. The file has to be opened to some "output" mode otherwise the command fails to print anything into the file. The command can also print into an opened socket (a file opened for mode socket). If the file is not opened then the expressions in the list `print_list` are not evaluated and the command actually does nothing. If the file is opened, but not for a kind of "output" mode then the expressions in the `print_list` are evaluated but the printing does not happen. Neither printing to a non-opened file number nor printing to a file opened for some read-only mode generates error.

If there is no `print_list` specified the command prints a new line. In other words if the keyword `PRINT` stands on the command with the optional `#` and the file number but without anything to print then the command will print a new line character.

Note that unlike other BASIC implementations the command `PRINT` does not accept print list formatters, like `AT` or semicolons and does not tabify the output. The arguments are printed to the file or to the standard output one after the other without any intrinsic space or tab added. Also the print statements does not print a new line at the end of the print list unless the new line character is explicitly defined or if there is no print list at all following the command.

This command prints the elements of the `print_list`. The argument `print_list` is a comma separated list of expressions. The expressions are evaluated one after the other and are printed to the standard output or to the file.

The command prints the `print_list` to an opened file given by the file number `fn`. If `fn` (along with the `#` character) is not specified the command prints to the standard output. The file has to be opened to some "output" mode otherwise the command fails to print

anything into the file. The command can also print into an opened socket (a file opened for mode socket). If the file is not opened then the expressions in the list `print_list` are not evaluated and the command actually does nothing. If the file is opened, but not for a kind of "output" mode then the expressions in the `print_list` are evaluated but the printing does not happen. Neither printing to a non-opened file number nor printing to a file opened for some read-only mode generates error.

If there is no `print_list` specified the command prints a new line. In other words if the keyword `PRINT` stands on the command with the optional `#` and the file number but without anything to print then the command will print a new line character.

Note that unlike other BASIC implementations the command `PRINT` does not accept print list formatters, like `AT` or semicolons and does not tabify the output. The arguments are printed to the file or to the standard output one after the other without any intrinsic space or tab added. Also the print statements does not print a new line at the end of the print list unless the new line character is explicitly defined or if there is no print list at all following the command.

This command prints the elements of the `print_list`. The argument `print_list` is a comma separated list of expressions. The expressions are evaluated one after the other and are printed to the standard output or to the file.

The command prints the `print_list` to an opened file given by the file number `fn`. If `fn` (along with the `#` character) is not specified the command prints to the standard output. The file has to be opened to some "output" mode otherwise the command fails to print anything into the file. The command can also print into an opened socket (a file opened for mode socket). If the file is not opened then the expressions in the list `print_list` are not evaluated and the command actually does nothing. If the file is opened, but not for a kind of "output" mode then the expressions in the `print_list` are evaluated but the printing does not happen. Neither printing to a non-opened file number nor printing to a file opened for some read-only mode generates error.

If there is no `print_list` specified the command prints a new line. In other words if the keyword `PRINT` stands on the command with the optional `#` and the file number but without anything to print then the command will print a new line character.

Note that unlike other BASIC implementations the command `PRINT` does not accept print list formatters, like `AT` or semicolons and does not tabify the output. The arguments are printed to the file or to the standard output one after the other without any intrinsic space or tab added. Also the print statements does not print a new line at the end of the print list unless the new line character is explicitly defined or if there is no print list at all following the command.

25.162 RANDOMIZE

Seed the random number generator. If the command is presented without argument the random number generator is seed with the actual time. If argument is provided the random number generator is seed with the argument following the keyword `RANDOMIZE`.

25.163 ref v1 = v2

Assign a variable to reference another variable. Following this command altering one of the variables alters both variables. In other words this command can be used to define a kind of alias to a variable. The mechanism is the same as local variable of a function is an alias of a variable passed to the function as actual argument. The difference is that this reference is not automatically released when some function returns, but rather it is alive so long as long the referencing variable is not undefined saying `undef variable` in a command.

To have an alias to a variable is not something of a great value though. It becomes a real player when the 'variable' is not just an ordinary 'named' variable but rather part of an array (or associative array). Using this mechanisms the programmer can build up arbitrary complex memory structures without caring such complex things as pointers for example in C. This is a simple BASIC way of building up complex memory structures.

25.164 REPEAT

This command implements a loop which is repeated so long as long the expression standing after the loop closing line `UNTIL` becomes `true`. The loop starts with a line containing the keyword `REPEAT` and finishes with the line `UNTIL expression`.

```
repeat
  ...
  commands to repeat
  ...
until expression
```

The expression is evaluated each time after the loop is executed. This means that the commands inside the loop are executed at least once.

This kind of loop syntax is not usual in BASIC dialects but can be found in languages like PASCAL. Implementing this loop in ScriptBasic helps those programmers, who have PASCAL experience.

See also See ⟨undefined⟩ [WHILE], page ⟨undefined⟩, See ⟨undefined⟩ [DOUNTIL], page ⟨undefined⟩, See ⟨undefined⟩ [DOWHILE], page ⟨undefined⟩, See ⟨undefined⟩ [REPEAT], page ⟨undefined⟩, See ⟨undefined⟩ [DO], page ⟨undefined⟩ and See ⟨undefined⟩ [FOR], page ⟨undefined⟩.

25.165 REPLACE(base_string,search_string,replace_string [,number_of_replaces] [,position])

This function replaces one or more occurrences of a sub-string in a string. `REPLACE(a,b,c)` searches the string `a` seeking for occurrences of sub-string `b` and replaces each of them with the string `c`.

The fourth and fifth arguments are optional. The fourth argument specifies the number of replaces to be performed. If this is missing or is `undef` then all occurrences of string `b` will be replaced. The fifth argument may specify the start position of the operation. For example the function call

```
REPLACE("alabama mama", "a","x",3,5)
```

will replace only three occurrences of string `"a"` starting at position 5. The result is `"alabxmx mxma"`.

25.166 RESET

This command closes all files opened by the current BASIC program. This command usually exists in most BASIC implementation. There is no need to close a file before a BASIC program finishes, because the interpreter automatically closes all files that were opened by the program.

This command closes all files opened by the current BASIC program. This command usually exists in most BASIC implementation. There is no need to close a file before a BASIC program finishes, because the interpreter automatically closes all files that were opened by the program.

This command closes all files opened by the current BASIC program. This command usually exists in most BASIC implementation. There is no need to close a file before a BASIC program finishes, because the interpreter automatically closes all files that were opened by the program.

25.167 RESET DIRECTORY [#] dn

Reset the directory file name list and start from the first file name when the next call to See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩ is performed.

See also See ⟨undefined⟩ [OPENDIR], page ⟨undefined⟩, See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩, See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩, See ⟨undefined⟩ [EOD], page ⟨undefined⟩.

Reset the directory file name list and start from the first file name when the next call to See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩ is performed.

See also See ⟨undefined⟩ [OPENDIR], page ⟨undefined⟩, See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩, See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩, See ⟨undefined⟩ [EOD], page ⟨undefined⟩.

Reset the directory file name list and start from the first file name when the next call to See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩ is performed.

See also See ⟨undefined⟩ [OPENDIR], page ⟨undefined⟩, See ⟨undefined⟩ [CLOSEDIR], page ⟨undefined⟩, See ⟨undefined⟩ [NEXTFILE], page ⟨undefined⟩, See ⟨undefined⟩ [EOD], page ⟨undefined⟩.

25.168 RESUME [label | next]

Resume the program execution after handling the error. `RESUME` without argument tries to execute the same line again that caused the error. `RESUME NEXT` tries to continue execution after the line that caused the error. `RESUME label` tries to continue execution at the specified label.

See also See ⟨undefined⟩ [ONERRORGOTO], page ⟨undefined⟩, See ⟨undefined⟩ [ONERRORRESUME], page ⟨undefined⟩ and See ⟨undefined⟩ [ERROR], page ⟨undefined⟩.

25.169 RETURN

Return from a subroutine started with See ⟨undefined⟩ [GOSUB], page ⟨undefined⟩. For more information see the documentation of the command See ⟨undefined⟩ [GOSUB], page ⟨undefined⟩.

25.170 REWIND [#]fn

Positions the file cursor to the start of the file. This is the same as `SEEK fn,0` or `SEEK #fn,0`

The argument to the statement is the file number used in the See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement to open the file. The character `#` is optional and can only be used for compatibility reasons. Positions the file cursor to the start of the file. This is the same as `SEEK fn,0` or `SEEK #fn,0`

The argument to the statement is the file number used in the See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement to open the file. The character `#` is optional and can only be used for compatibility reasons. Positions the file cursor to the start of the file. This is the same as `SEEK fn,0` or `SEEK #fn,0`

The argument to the statement is the file number used in the See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement to open the file. The character `#` is optional and can only be used for compatibility reasons.

25.171 RIGHT(string,len)

Creates the right of a string. The first argument is the string. The second argument is the number of characters that should be put into the result. If this value is larger than the number of characters in the string then all the string is returned.

See also See ⟨undefined⟩ [MID], page ⟨undefined⟩, See ⟨undefined⟩ [LEFT], page ⟨undefined⟩.

25.171.1 RIGHT Details

`RIGHT(x,y)` cuts out a substring of `y` characters from the right of the string `x`. If the first argument is not defined the result is also `undef`. Otherwise the first argument is converted to string and the second argument is converted to integer value.

If the second argument is not defined or has negative value it is treated as numeric zero and as such the result string will be empty string.

For compatibility reasons you can append a dollar (`$`) sign to the end of the function identifier.

25.172 RND

Returns a random number as generated by the C function `rand()`. Note that this random number generator usually provided by the libraries implemented for the C compiler or the operating system is not the best quality ones. If you need really good random number generator then you have to use some other libraries that implement reliable RND functions.

25.173 ROUND

This function rounds its argument. The first argument is the number to round, and the optional second argument specifies the number of fractional digits to round to.

The function rounds to integer value if the second argument is missing.

The return value is long if the number of decimal places to keep is zero, otherwise the return value is double.

Negative value for the number of decimal places results rounding to integer value.

`ROUND(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

Also `ROUND(val,undef)` is equivalent to `ROUND(value)`.

See also See ⟨undefined⟩ [INT], page ⟨undefined⟩, See ⟨undefined⟩ [FRAC], page ⟨undefined⟩ and See ⟨undefined⟩ [FIX], page ⟨undefined⟩

25.174 RTRIM()

Remove the space from the right of the string.

25.175 SEC

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the seconds value of that time. If the argument is missing the function uses the actual local time.

25.176 SECANT

This is a planned function to calculate the secant of the argument.

25.177 SEEK fn,position

Go to a specified position in an open file. You can use this command to position the file pointer to a specific position. The next read or write operation performed on the file will be performed on that very position that was set using the command `SEEK`. The first argument is the file number that was used in the statement `OPEN` to open the file. The second argument is the position where the file pointer is to be set.

The position is counted from the start of the file counting records. The actual file pointer will be set **after>** the record `position`. This means that if for example you want to set the file pointer To the start of the file then you have to `SEEK fn,0`. This will set the File pointer before the first record.

If there was no record length specified when the file was opened the counting takes bytes. There is no special "record" structure of a file as it is usually under UNIX or Windows NT. The record is merely the number of bytes treated as a single unit specified during file opening. Go to a specified position in an open file. You can use this command to position the file pointer to a specific position. The next read or write operation performed on the file will be performed on that very position that was set using the command `SEEK`. The first argument is the file number that was used in the statement `OPEN` to open the file. The second argument is the position where the file pointer is to be set.

The position is counted from the start of the file counting records. The actual file pointer will be set **after>** the record `position`. This means that if for example you want to set the file pointer To the start of the file then you have to `SEEK fn,0`. This will set the File pointer before the first record.

If there was no record length specified when the file was opened the counting takes bytes. There is no special "record" structure of a file as it is usually under UNIX or Windows NT. The record is merely the number of bytes treated as a single unit specified during file opening. Go to a specified position in an open file. You can use this command to position the file pointer to a specific position. The next read or write operation performed on the file will be performed on that very position that was set using the command `SEEK`. The first argument is the file number that was used in the statement `OPEN` to open the file. The second argument is the position where the file pointer is to be set.

The position is counted from the start of the file counting records. The actual file pointer will be set **after>** the record `position`. This means that if for example you want to set the file pointer To the start of the file then you have to `SEEK fn,0`. This will set the File pointer before the first record.

If there was no record length specified when the file was opened the counting takes bytes. There is no special "record" structure of a file as it is usually under UNIX or Windows NT. The record is merely the number of bytes treated as a single unit specified during file opening.

25.178 SET FILE filename parameter=value

Set some of the parameters of a file. The parameter can be:

`owner` set the owner of the file. This operation requires `root` permission on UNIX or `Administrator` privileges on Windows NT. The value should be the string representation of the UNIX user or the Windows NT domain user.

`createtime`

`modifytime`

`accesstime`

Set the time of the file. The value should be the file time in seconds since January 1,1970. 00:00GMT.

If the command can not be executed an error is raised. Note that setting the file owner also depends on the file system. For example FAT file system does not store the owner of a file and thus can not be set.

Also setting the file time on some file system may be unsuccessful for values that are successful under other file systems. This is because different file systems store the file times using different possible start and end dates and resolution. For example you can set a file to hold the creation time to be January 1, 1970 0:00 under NTFS, but not under FAT.

The different file systems store the file times with different precision. Thus the actual time set will be the closest time not later than the specified in the command argument. For this reason the values returned by the functions `File***Time` may not be the same that was specified in the `SET FILE` command argument.

Set some of the parameters of a file. The parameter can be:

`owner` set the owner of the file. This operation requires `root` permission on UNIX or `Administrator` privileges on Windows NT. The value should be the string representation of the UNIX user or the Windows NT domain user.

`createtime`

`modifytime`

`accesstime`

Set the time of the file. The value should be the file time in seconds since January 1,1970. 00:00GMT.

If the command can not be executed an error is raised. Note that setting the file owner also depends on the file system. For example FAT file system does not store the owner of a file and thus can not be set.

Also setting the file time on some file system may be unsuccessful for values that are successful under other file systems. This is because different file systems store the file times using different possible start and end dates and resolution. For example you can set a file to hold the creation time to be January 1, 1970 0:00 under NTFS, but not under FAT.

The different file systems store the file times with different precision. Thus the actual time set will be the closest time not later than the specified in the command argument. For this reason the values returned by the functions `File***Time` may not be the same that was specified in the `SET FILE` command argument.

Set some of the parameters of a file. The parameter can be:

`owner` set the owner of the file. This operation requires `root` permission on UNIX or `Administrator` privileges on Windows NT. The value should be the string representation of the UNIX user or the Windows NT domain user.

`createtime`

`modifytime`

`accesstime`

Set the time of the file. The value should be the file time in seconds since January 1,1970. 00:00GMT.

If the command can not be executed an error is raised. Note that setting the file owner also depends on the file system. For example FAT file system does not store the owner of a file and thus can not be set.

Also setting the file time on some file system may be unsuccessful for values that are successful under other file systems. This is because different file systems store the file times using different possible start and end dates and resolution. For example you can set a file to hold the creation time to be January 1, 1970 0:00 under NTFS, but not under FAT.

The different file systems store the file times with different precision. Thus the actual time set will be the closest time not later than the specified in the command argument. For this reason the values returned by the functions `File***Time` may not be the same that was specified in the `SET FILE` command argument.

25.179 SET JOKER "c" TO "abcdefgh..."

Set a joker character to match certain characters when using the See ⟨undefined⟩ [LIKE], page ⟨undefined⟩ operator. The joker character `"c"` can be one of the following characters

```
*  #  $    ?  &  %  !  +  /  |  <  >
```

The string after the keyword `TO` should contain all the characters that the joker character should match. To have the character to match only itself to be a normal character say

```
SET NO JOKER "c"
```

See also See ⟨undefined⟩ [SETWILD], page ⟨undefined⟩, See ⟨undefined⟩ [LIKE], page ⟨undefined⟩ (details), See ⟨undefined⟩ [JOKER], page ⟨undefined⟩

25.180 SET WILD "c" TO "abcdefgh..."

Set a wild character to match certain characters when using the See ⟨undefined⟩ [LIKE], page ⟨undefined⟩ operator. The wild character `"c"` can be one of the following characters

```
*  #  $    ?  &  %  !  +  /  |  <  >
```

The string after the keyword `TO` should contain all the characters that the wild card character should match. To have the character to match only itself to be a normal character say

```
SET NO WILD "c"
```

See also See ⟨undefined⟩ [SETJOKER], page ⟨undefined⟩, See ⟨undefined⟩ [LIKE], page ⟨undefined⟩ (details), See ⟨undefined⟩ [JOKER], page ⟨undefined⟩

25.181 SIN

Calculates the sine of the argument. If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`SIN(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.182 SLEEP(n)

Suspend the execution of the interpreter (process or thread) for `n` seconds.

Whenever the program has to wait for a few seconds it is a good idea to call this function. Older BASIC programs originally designed for old personal computers like Atari, Amiga, ZX Spectrum intend to use empty loop to wait time to elapse. On modern computers this is a bad idea and should not be done.

If you execute an empty loop to wait you consume CPU. Because the program does not access any resource to wait for it actually consumes all the CPU time slots that are available. This means that the computer slows down, does not respond to user actions timely.

Different computers run with different speed and an empty loop consuming 20sec on one machine may run 2 minutes on the other or just 10 millisec. You can not reliably tell how much time there will be during the empty loop runs.

When you call `SLEEP(n)` the operating system is called telling it that the code does not need the CPU for `n` seconds. During this time the program is suspended and the operating system executes other programs as needed. The code is guaranteed to return from the function `SLEEP` not sooner than `n` seconds, but usually it does return before the second `n+1` starts.

Suspend the execution of the interpreter (process or thread) for `n` seconds.

Whenever the program has to wait for a few seconds it is a good idea to execute this command. Older BASIC programs originally designed for old personal computers like Atari, Amiga, ZX Spectrum intend to use empty loop to wait time to elapse. On modern computers this is a bad idea and should not be done.

If you execute an empty loop to wait you consume CPU. Because the program does not access any resource to wait for it actually consumes all the CPU time slots that are available. This means that the computer slows down, does not respond to user actions timely.

Different computers run with different speed and an empty loop consuming 20sec on one machine may run 2 minutes on the other or just 10 millisec. You can not reliably tell how much time there will be during the empty loop runs.

When you execute `SLEEP n` the operating system is called telling it that the code does not need the CPU for `n` seconds. During this time the program is suspended and the operating system executes other programs as needed. The code is guaranteed to return from the function `SLEEP` not sooner than `n` seconds, but usually it does return before the second `n+1` starts.

Suspend the execution of the interpreter (process or thread) for `n` seconds.

Whenever the program has to wait for a few seconds it is a good idea to call this function. Older BASIC programs originally designed for old personal computers like Atari, Amiga, ZX Spectrum intend to use empty loop to wait time to elapse. On modern computers this is a bad idea and should not be done.

If you execute an empty loop to wait you consume CPU. Because the program does not access any resource to wait for it actually consumes all the CPU time slots that are

available. This means that the computer slows down, does not respond to user actions timely.

Different computers run with different speed and an empty loop consuming 20sec on one machine may run 2 minutes on the other or just 10 millisec. You can not reliably tell how much time there will be during the empty loop runs.

When you call `SLEEP(n)` the operating system is called telling it that the code does not need the CPU for `n` seconds. During this time the program is suspended and the operating system executes other programs as needed. The code is guaranteed to return from the function `SLEEP` not sooner than `n` seconds, but usually it does return before the second `n+1` starts.

25.183 SPACE(n)

Return a string of length `n` containing spaces.

25.184 SPLIT string BY string TO var_1,var_2,var_3,...,var_n

Takes the string and splits into the variables using the second string as delimiter.

25.185 SPLITA string BY string TO array

Split a string into an array using the second string as delimiter. If the string has zero length the array becomes undefined. When the delimiter is zero length string each array element will contain a single character of the string.

See also See ⟨undefined⟩ [SPLIT], page ⟨undefined⟩

25.186 SPLITAQ string BY string QUOTE string TO array

Split a string into an array using the second string as delimiter. The delimited fields may optionally be quoted with the third string. If the string to be split has zero length the array becomes undefined. When the delimiter is a zero length string each array element will contain a single character of the string.

Leading and trailing delimiters are accepted and return an empty element in the array. For example :-

```
SPLITAQ ",'A,B',C," BY "," QUOTE "'" TO Result
```

will generate

```
Result[0] = ""
Result[1] = "A,B"
Result[2] = "C"
Result[3] = ""
```

Note that this kind of handling of trailing and leading empty elements is different from the handling of the same by the command See ⟨undefined⟩ [SPLIT], page ⟨undefined⟩ and

See ⟨undefined⟩ [SPLITA], page ⟨undefined⟩ which do ignore those empty elements. This command is useful to handle lines exported as CSV from Excel or similar application.

The QUOTE string is really a string and need not be a single character. If there is an unmatched quote string in the string to be split then the rest of the string until its end is considered quoted.

If there is an unmatched

See also See ⟨undefined⟩ [SPLITA], page ⟨undefined⟩

This command was suggested and implemented by Andrew Kingwell (`Andrew.Kingwell@idstelecom.co.uk`)

25.187 SQR

Calculates the square root of the argument.

If the result is within the range of an integer value on the actual architecture then the result is returned as an integer, otherwise it is returned as a real value.

`SQR(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

If the argument is a negative number the result of the function is `undef` or the function raises error if the option `RaiseMathError` has the bit `sbMathErrDiv` set.

If the square root of the argument is an integer number then the function returns an integer number. In other cases the returned value is real even if the argument itself is integer.

Note that this function has the opposite meaning in the language PASCAL, namely the square of the number. This may cause some problem if you are experienced in PASCAL programming. In that language `SQRT` notes the square *root*> of a number.

25.188 STOP

This command stops program execution. There is no possibility to restart program execution after this command was executed.

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

25.189 STR(n)

Converts a number to string. This function is rarely needed, because conversion is done automatically.

25.189.1 STR Details

Converts a number to string. This function is rarely needed, because conversion is done automatically. However you may need

```
STRING(13,STR(a))
```

to be sure that the value `a` is interpreted as string value.

25.190 STRING(n,code)

Create a string of length `n` containing characters `code`. If `code` is a string then the first character of the string is used to fill the result. Otherwise `code` is converted to long and the ASCII code is used.

25.191 STRREVERSE(string)

Return the reversed string (aka. all the characters in the string in reverse order).

25.192 SUB fun()

This command should be used to define a subroutine. A subroutine is a piece of code that can be called by the BASIC program from the main part or from a function or subroutine.

```
SUB sub(a,b,c)
...
END SUB
```

The end of the subroutine is defined by the line containing the keywords `END SUB`.

Note that functions and subroutines are not really different in ScriptBasic. ScriptBasic allows you to return a value from a subroutine and to call a function using the command `CALL`. It is just a convention to have separately `SUB` and `FUNCTION` declarations.

For detailed information please read the documentation of the command See ⟨undefined⟩ [FUNCTION], page ⟨undefined⟩

25.193 swap a,b

Planned command.

This command swaps two variables.

25.194 SYSTEM(executable_program)

This function should be used to start an external program in a separate process in asynchronous mode. In other words you can start a process and let it run by itself and not wait for the process to finish. After starting the new process the BASIC program goes on parallel with the started external program.

The return value of the function is the PID of the newly created process.

This function should be used to start an external program in a separate process in asynchronous mode. In other words you can start a process and let it run by itself and not wait for the process to finish. After starting the new process the BASIC program goes on parallel with the started external program.

The return value of the function is the PID of the newly created process.

If the program specified by the argument can not be started then the return value is zero. Under UNIX the program may return a valid PID even in this case. This is because

UNIX first makes a copy of the process that wants to start another and then replaces the new process image with the program image to be started. In this case the new process is created and the command `SYSTEM` has no information on the fact that the new process was not able to replace the executable image of itself. In this case, however, the child process has a very short life.

This function should be used to start an external program in a separate process in asynchronous mode. In other words you can start a process and let it run by itself and not wait for the process to finish. After starting the new process the BASIC program goes on parallel with the started external program.

The return value of the function is the PID of the newly created process.

25.195 TAN

This is a planned function to calculate the tangent of the argument.

25.196 TAN2

This is a planned function to calculate the tangent of the ratio of the two arguments.

25.197 TEXTMODE [# fn] | input | output

Set an opened file handling to text mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also See ⟨undefined⟩ [BINMODE], page ⟨undefined⟩ Set an opened file handling to text mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also See ⟨undefined⟩ [BINMODE], page ⟨undefined⟩ Set an opened file handling to text mode.

The argument is either a file number with which the file was opened or one of keywords `input` and `output`. In the latter case the standard input or output is set.

See also See ⟨undefined⟩ [BINMODE], page ⟨undefined⟩

25.198 TIMEVALUE

This function gets zero or more, at most six arguments and interprets them as year, month, day, hour, minute and seconds and calculates the number of seconds elapsed since January 1, 1970 till the time specified. If some arguments are missing or `undef` the default values are the following:

year = 1970

month = January

day = 1st
hours = 0
minutes = 0
seconds = 0

25.199 TRIM()

Remove the space from both ends of the string.

25.200 TRUE

This built-in constant is implemented as an argument less function. Returns the value `true`.

25.201 TRUNCATE fn,new_length

Truncate an opened file to the specified size. The first argument Has to be the file number used in the See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement opening the file. The second argument is the number of records to be in the file after it is truncated.

The size of a record has to be specified when the file is opened. If the size Of a record is not specified in number of bytes then the command `TRUNCATE` Does truncate the file to the number of specified bytes instead of records. (In other words the record length is one byte.)

When the file is actually shorter than the length specified by the command argument the command `TRUNCATE` automatically extends the file padding with bytes containing the value 0.

Truncate an opened file to the specified size. The first argument Has to be the file number used in the See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement opening the file. The second argument is the number of records to be in the file after it is truncated.

The size of a record has to be specified when the file is opened. If the size Of a record is not specified in number of bytes then the command `TRUNCATE` Does truncate the file to the number of specified bytes instead of records. (In other words the record length is one byte.)

When the file is actually shorter than the length specified by the command argument the command `TRUNCATE` automatically extends the file padding with bytes containing the value 0.

Truncate an opened file to the specified size. The first argument Has to be the file number used in the See ⟨undefined⟩ [OPEN], page ⟨undefined⟩ statement opening the file. The second argument is the number of records to be in the file after it is truncated.

The size of a record has to be specified when the file is opened. If the size Of a record is not specified in number of bytes then the command `TRUNCATE` Does truncate the file to the number of specified bytes instead of records. (In other words the record length is one byte.)

When the file is actually shorter than the length specified by the command argument the command `TRUNCATE` automatically extends the file padding with bytes containing the value 0.

25.202 TYPE

This function can be used to determine the type of an expression. The function returns a numeric value that describes the type of the argument. Although the numeric values are guaranteed to be the one defined here it is recommended that you use the predefined symbolic constant values to compare the return value of the function against. The function return value is the following

`SbTypeUndef` 0 if the argument is `undef`.

`SbTypeString` 1 if the argument is string.

`SbTypeReal` 2 if the argument is real.

`SbTypeInteger` 3 if the argument is integer.

`SbTypeArray` 4 if the argument is an array.

See also See ⟨undefined⟩ [ISARRAY], page ⟨undefined⟩, See ⟨undefined⟩ [ISSTRING], page ⟨undefined⟩, See ⟨undefined⟩ [ISINTEGER], page ⟨undefined⟩, See ⟨undefined⟩ [ISREAL], page ⟨undefined⟩, See ⟨undefined⟩ [ISNUMERIC], page ⟨undefined⟩, See ⟨undefined⟩ [ISDEFINED], page ⟨undefined⟩, See ⟨undefined⟩ [ISUNDEF], page ⟨undefined⟩, See ⟨undefined⟩ [ISEMPTY], page ⟨undefined⟩.

25.203 UBOUND

This function can be used to determine the highest occupied index of an array. Note that arrays are increased in addressable indices automatically, thus it is not an error to use a higher index that the value returned by the function `UBOUND`. On the other hand all the element having index larger than the returned value are `undef`.

The argument of this function has to be an array. If the argument is an ordinary value, or a variable that is not an array the value returned by the function will be `undef`.

`UBOUND(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

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

25.204 UCASE()

Uppercase a string.

25.205 UNDEF variable

Sets the value of a variable (or some other ScriptBasic left value) to be undefined. This command can also be used to release the memory that was occupied by an array when the variable holding the array is set to `undef`.

When this command is used as a function (with or without, but usually without parentheses), it simply returns the value `undef`.

25.205.1 UNDEF Details

Note that when this command is called in a function then the local variable is undefined and the caller variable passed by reference is not changed. Therefore

```
sub xx(a)
 undef a
end sub

q = 1
xx q
print q
```

will print 1 and not `undef`.

On the other hand

```
sub xx(a)
 a = undef
end sub

q = 1
xx q
print q
```

does print `undef`.

25.206 UNPACK string BY format TO v1,v2,...,vn

Unpack the binary string `string` using the format string into the variables. The format string should have the same format as the format string the in the function See ⟨undefined⟩ [PACK], page ⟨undefined⟩.

25.207 VAL

Converts a string to numeric value. If the string is integer it returns an integer value. If the string contains a number presentation which is a float number the returned value is real. In case the argument is already numeric no conversion is done.

`VAL(undef)` is `undef` or raises an error if the option `RaiseMatherror` is set in bit `sbMathErrUndef`.

25.208 WAITPID(PID,ExitCode)

This function should be used to test for the existence of a process.

The return value of the function is 0 if the process is still running. If the process has exited (or failed in some way) the return value is `1` and the exit code of the process is stored in `ExitCode`.

25.209 WEEKDAY

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the week day value of that time. If the argument is missing the function uses the actual local time. In other words it returns what day it is at the moment.

25.210 WHILE condition

Implements the 'while' loop as it is usually done in most basic implementations. The loop starts with the command `while` and finished with the line containing the keyword `wend`. The keyword `while` is followed by an expression and the loop is executes so long as long the expression is evaluated `true`.

```
while expression
  ...
  commands to repeat
  ...
wend
```

The expression is evaluated when the looping starts and each time the loop is restarted. It means that the code between the `while` and `wend` lines may be skipped totally if the expression evaluates to some `false` value during the first evaluation before the execution starts the loop.

In case some condition makes it necessary to exit the loop from its middle then the command See ⟨undefined⟩ [GOTO], page ⟨undefined⟩ can be used.

ScriptBasic implements several looping constructs to be compatible with different BASIC language dialects. Some constructs are even totally interchangeable to let programmers with different BASIC experience use the one that fit they the best. See also See ⟨undefined⟩ [WHILE], page ⟨undefined⟩, See ⟨undefined⟩ [DOUNTIL], page ⟨undefined⟩, See ⟨undefined⟩ [DOWHILE], page ⟨undefined⟩, See ⟨undefined⟩ [REPEAT], page ⟨undefined⟩, See ⟨undefined⟩ [DO], page ⟨undefined⟩ and See ⟨undefined⟩ [FOR], page ⟨undefined⟩.

25.211 YEAR

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the year value of that time. If the argument is missing it uses the actual local time to calculate the year value. In other words it returns the actual year.

25.212 YEARDAY

This function accepts one argument that should express the time in number of seconds since January 1, 1970 0:00 am and returns the year-day value of that time. This is actually the number of the day inside the year so that January 1st is #1 and December 31 is #365 (or 366 in leap years). If the argument is missing the function uses the actual local time.