TKS

-- a portable, JIT accelerated script-engine for glue'ing C and C++ frameworks.

Kirchbachstrasse 195
D-28211 Bremen

E-Mail : <bs@tkscript.de>

Web : http://tkscript.de

Tel : +0049- (0)421 - 577 16 94

last updated: 30. December 2003

table of contents

1. Introduction.....5

2. Background.....5

3. System requirements and installation.....6
3.1. Compiling the TKS sources under Unix.....7
3.2. Compiling the TKSDL sources under Unix.....8

4. License.....9

5. Invoking the scriptengine.....10
5.1. Command line arguments.....11

6. Projects, the virtual filesystem and TKX archives.....12
6.1. Accessing local files.....12
6.2. Script source files.....13
6.3. Creating packaged TKX archives.....13
6.4. Publishing TKX archives.....13

7. Modules and comments.....14
7.1. Declaration of modules.....14
7.2. Module archives.....14
7.3. Scope of classes and constants.....14
7.4. Scope of functions.....14
7.5. The Main module.....15
7.6. Program code outside of functions.....15
7.7. Comments.....15

8. Identifiers, keywords, literals and constants.....16
8.1. Escape sequences.....16
8.2. Identifiers.....16
8.3. Delimiters and operator keywords.....17
8.4. Reserved keywords.....17
8.5. Literals.....18
8.6. User defined constants.....19

9. Datatypes and variable declarations.....20
9.1. Core API datatypes.....20
9.2. Variable declarations.....21

10. Objectreferences (pointers).....22
10.1. Typed pointers.....22
10.2. Untyped pointers.....22
10.3. Deleting a pointer.....23
10.4. Pointer arguments.....23

11. Strings 24
11.1. Stringlists.....25
11.2. Formatted Textfiles.....26

table of contents *(cont.)*

12. Operators.....27
12.1. Operator classes and priorities.....27
12.2. Assignment operators.....27
12.3. Unary operators.....28
12.4. The ternary operator.....29
12.5. The streaming << operator.....29

13. Expressions.....30
13.1. the "range" expression.....31
13.2. the "mini-if" ternary expression 31
13.3. the "new" expression 31

14. Statements and flow control.....32
14.1. the "if" statement.....33
14.2. the "do..while" statement.....34
14.3. the "while" statement.....34
14.4. the "switch" statement.....35
14.5. the "for" statement.....36
14.6. the "loop" statement.....37
14.7. the "foreach" statement.....38
14.8. the "prepare" statement.....39
14.9. the "clamp" and "wrap" statements.....39
14.10. the "use" statement.....40
14.11. the "define" statement.....41
14.12. the "enum" statement.....41
14.13. the "function" statement 41
14.14. the "class" statement.....41

15. Functions.....42
15.1. User defined functions.....42
15.1.1. Forward declarations.....42
15.1.2. Functions in other modules.....43
15.1.3. Variable return types.....43
15.2. TKS API functions.....44

16. Complex data structures
16.1. Classes.....45
16.1.1. Members.....45
16.1.2. Script vs. API classes.....45
16.1.3. Methods.....46
16.1.4. Constructors and Destructors.....47
16.1.5. Inheritance and late binding.....48
16.2. Arrays and Arraylists.....50
16.3. Associative Arrays (HashTables).....51
16.4. Array initializers.....51
16.5. Multidimensional Arrays.....52
16.5.1. Hash of hashes.....52
16.6. Pools.....53
16.7. Trees.....54
16.8. Stacks.....54

table of contents *(cont.)*

17. the Just-In-Time (*JIT*) Compiler.....55
17.1. Usage.....56
17.2. Limitations and hints.....58
17.3. P-Code instruction set.....**59**

18. the *YAC* C/C++ plugin interface.....65
18.1. Key features.....65
18.2. Basic datastructures.....65
18.3. The "YAC_Host" interface.....67
18.4. Functions.....68
18.5. Classes.....69
18.5.1. C++ Example.....69
18.5.2. Constants and macros.....70
18.5.3. Object templates.....71
18.5.4. The YAC_Object class.....72
18.6. Constants.....74
18.7. Properties via set/get.....74
18.8. Property classes.....
18.9. Signals.....
18.10. Streams.....
18.11. Serialization.....
18.12. Arrays.....
18.13. Iterators.....
18.14. Calling script functions.....
18.15. Creating and deleting Objects.....

19. API Reference.....81
19.1. Buffer.....82
19.2. ClassArray.....83
19.3. Configuration.....84
19.4. Envelope.....85
19.5. Event.....86
19.6. File.....87
19.7. Float.....88
19.8. FloatArray.....89
19.9. HashTable.....90
19.10. IntArray.....91
19.11. Integer.....92
19.12. ObjectArray.....93
19.13. PointerArray.....94
19.14. Pool.....95
19.15. Stack.....96
19.16. Stream.....97
19.17. String.....98
19.18. StringArray.....99
19.19. StringIterator.....100
19.20. Time.....101
19.21. TKS.....102
19.22. TreeNode.....103

20. Links.....104

Appendix A. (ASCII table).....105

1. Introduction

TKS, short for "*toolkit script-language*", is an open and portable glue[1] language for native code libraries. The script engine focuses on C++ API backends but also allows for C and Assembler functions. The scripts look much like C/C++/JavaScript/PHP so it will not take someone too long to get familiar with the syntax. A CPU table based (i.e. rather portable) *Just In Time* (JIT) compiler can speed up script execution by up to 100 times.

TKS serves as a scriptable application host for custom C++ libraries ("*plugins*"). Bindings for OpenGL,SDL,SDL_net,libpng,zlib,MiniFMOD and MySQL are available while other extensions are currently being developed, e.g. for the FOX GUI toolkit.

A TKS application project can be compiled to a gzip compressed TKX archive which will contain all data necessary to run that application. This technique makes it possible to deploy applications without prior local installation (besides the TKS runtime) which suggests the use for network based systems, e.g. the world wide web or intranets. TKX archives, in contrary to regular *executables,* are hardware-independent and can thus be used across multiple platforms.

The current implementation of the scriptlanguage, its source codes as well as precompiled distributions are available under the terms of the **GNU General Public License**.

The interface for application-specific C/C++ extension libraries (YAC) is released under terms of the **GNU Lesser General Public License** which also allows for proprietary (i.e. *closed source)* extensions (*plugins*).

2. Background

In the long run, this project is meant to ease development of every day user interfaces / applications and tools while providing an open and stable platform that combines traditional system programming languages like C with a "highlevel" scripting language.

This mixture makes it possible to, on the one hand remove application logic from native code so the code becomes reusable and on the other hand add further abstraction layers to existing code libraries. These abstraction layers may be implemented as a script module, e.g. for a generic database interface, or come in form of a plugin, e.g. a binding for a 3D engine or GUI library which provides an entire new application framework.

The TKS API classes itself only provide basic tools for arrays/hashtables/strings and the like.

The TKSDL plugin is an (stable) example of binding a multimedia application framework to TKS. The plugin extends the script API by classes and functions for 3D acceleration, window and screenhandling, input devices, networking, sound, vectors and matrices and texture loading/saving.

1 am. "glue". Generic term for any interface logic or protocol that connects two component blocks. [jargon.8hz.com]

3. System requirements and installation

TKS is available either as a sourcecode distribution licensed under terms of the GNU General Public License- or as a precompiled binary-package.

The sources can be compiled under Win32, Linux, Cygwin (tested) as well as other Un*x compatible operating systems (Solaris, Irix, FreeBSD, ...).

Binaries are currently available for Win32 (Linux users should better use the source).

Precompiled binaries:
http://tkscript.de/files/tks-setup.exe
http://tkscript.de/files/tks-gcc3-shared-bin.tgz

Sources:
http://tkscript.de/files/tks-source.zip
http://tkscript.de/files/yac.zip
http://tkscript.de/plugins/tksdl.zip
http://tkscript.de/plugins/mysql.zip
http://tkscript.de/plugins/minifmod.zip

Examples (executable archives):
http://tkscript.de/tkx/sieve.tkx

Examples (executable archives, display hacks):
http://tkscript.de/tkx/bobfield.tkx
http://tkscript.de/tkx/crparticles.tkx
http://tkscript.de/tkx/juliaattractor.tkx
http://tkscript.de/tkx/retrovaders2.tkx
http://tkscript.de/tkx/torus.tkx

Examples (sources):
http://tkscript.de/tkp/tks-examples.zip
http://tkscript.de/tkp/retrovaders2.zip

Miscellaneous (e.g. Emacs "ini" file and more examples):
http://tkscript.de/downloads.html

The latest release of this documentation:
http://tkscript.de/files/tkscript.pdf

A step-by-step guide to compiling the sources can be found at http://tkscript.de/compiling.html and/or on the following page.

The *changelog* is available at http://tkscript.de/news.html.

3.1. Compiling the TKS sources under Unix

1. make sure you have `gcc`, `g++` and `gmake` (or a compatible make) installed
2. make sure you have zlib-devel and zlib installed; if you don't want zlib support you can disable it in `tks-config.h` (disable `DX_Z` define)
3. make sure you have the `unzip` command installed (or a similar tool to depack the source code archive)
4. get `tks-source.zip` or `tks-0.5.2.2-src.tar.gz`, e.g.

```
[root@wizball root]# mkdir tkscript; cd tkscript && curl http://tkscript.de/files/tks-0.5.2.2-
src.tar.gz | tar xfz -
  % Total    % Received % Xferd  Average Speed          Time             Curr.
                                 Dload  Upload Total    Current  Left    Speed
100 1079k  100 1079k    0     0   169k      0  0:00:06  0:00:06  0:00:00  172k

[root@wizball tkscript]# unzip files/tks-source.zip
Archive:  files/tks-source.zip
   creating: tks-source/
   .
   .

[root@wizball tkscript]# unzip files/yac.zip
   creating: yac/
  inflating: yac/yac.h
  inflating: yac/yac_host.h
  inflating: yac/tks-opengl.h
  inflating: yac/yac_createcommandlist2.h
  inflating: yac/file_id.diz
  inflating: yac/yac_createcommandlist1.h
  inflating: yac/tks-list.h
  inflating: yac/YAC-License.txt
  inflating: yac/yac_createcommands.h
  inflating: yac/yac_createcommands1.h
  inflating: yac/yac_createcommands2.h

[root@wizball tkscript]# unzip plugins/tksdl.zip
Archive:  plugins/tksdl.zip
   creating: tksdl/
  inflating: tksdl/COPYING
  .
  .
  .
```

5. change to the new `tks-source` directory

```
[root@wizball tkscript]# cd tks-source
[root@wizball tks-source]#
```

6. type

```
[root@wizball tks-source]# make -f makefile.linux shared
```

7. if everything has compiled successfully you now have a "tks" executable in the tks-source directory
8. you may want to test whether tks works by executing

```
[root@wizball tks-source]# ./tks sieve.tkp
      or

[root@wizball tks-source]# ./tks sieve.tkx
      if you compiled with zlib support
```

9. finally you may want to install it, e.g.

```
[root@wizball tks-source]# cp tks /usr/bin/tks
```

3.2. Compiling the TKSDL sources under Unix

1. make sure you have the `libGL`, `libGL-devel` OpenGL libraries and includes installed (usually `/usr/include/GL/`)
2. ..if you can't find the headerfiles..try `http://tkscript.de/files/opengl_includes.zip`
3. make sure you have `libGLU`, `libGLU-devel` installed
4. make sure you have libpng, libpng-devel installed
5. make sure you have zlib, zlib-devel installed
6. make sure you have SDL, SDL-devel installed
7. make sure you have SDL_net installed
8. get `yac.zip` and `tksdl.zip`
9. unzip `yac.zip` and `tksdl.zip` so the directory structure looks like this:

```
[root@wizball tkscript]# ls
tksdl/  tks-source/  yac/
[root@wizball tkscript]#
```

10. now change to the tksdl directory

```
[root@wizball tkscript]# cd tksdl
[root@wizball tksdl]#
```

11. type

```
[root@wizball tksdl]# make -f makefile.linux plugin
```

to compile the tksdl.so library

12. create a plugin directory on your system, e.g.

```
[root@wizball tksdl]# mkdir /usr/local/tks/plugins
```

13. create a module directory on your system, e.g.

```
[root@wizball tksdl]# mkdir /usr/local/tks/modules
```

14. type

```
[root@wizball tksdl]# cp tksdl.so /usr/local/tks/plugins/
```

to install the tksdl plugin

15. now you have to set an environment variable so that the tks plugin loader will be able to locate a requested plugin:

```
[root@wizball tksdl]# export TKS_PLUGIN_PATH=/usr/local/tks/plugins/
```

(don't forget the last slash!)

16. in order to include (reusable) script modules when creating new tks applications you'll need to set another env. var.:

```
[root@wizball tksdl]# export TKS_MODULE_PATH=/usr/local/tks/modules/
```

17. download an example tkx file, e.g.

```
[root@wizball tksdl]# curl http://tkscript.de/tkx/bobfield.tkx >bobfield.tkx
```

18. start the example file by typing

```
[root@wizball tksdl]# tks bobfield
```

19. if something goes wrong you may want to turn on some debug messages

```
[root@wizball tksdl]# tks -d 85 bobfield
```

20. .
21. .
22. caveat: on some linux systems the plugin loader is unable to locate the plugin regardless whether the tksdl.so is located in the same directory as the tks.exe or not; in this case you may also want to try

```
[root@wizball tks-source]# ./run.sh bobfield
```

(assuming that you have put the `tksdl.so` file in the `tks-source` directory along with the `bobfield.tkx` file)

23. in case you don't want to install or copy anything at all just try:

```
[root@wizball tks-source]# ./tks -pp ../tksdl/ bobfield
```

4. License

The TKS scriptengine and the TKSDL extension are released under the terms of the GNU Public License, please see http://www.gnu.org/licenses/licenses.html#GPL.

The extension interface ("YAC") is released under the terms of the Library GNU Public License, please see http://www.gnu.org/licenses/licenses.html#LGPL.

5. Invoking the scriptengine

After succesful installation of the TKS package, the scriptengine can be executed by issueing

```
$ /path/to/tks <sourcefile[.tkx]>
```

on a shell or by clicking on a TKX[1] file in a file resp. web browser.

A MIME handler for the TKX datatype is automatically registered by the tks-setup.exe installer.

The TKS executable contains both runtime environment and compiler for script applications. A separate compiler run, which is required by real programming languages like C++, Java and C# is not needed. This makes it possible to run an application on different platforms without repackaging the distributable archive.

To take your first steps with TKS you may create a text-file called test.tks containing

```
loop (10) trace "tks says hello, world.";
```

and execute it by issueing

```
$ tks test
```

or

```
$ tks test.tks
```

1the portable “executable“ format used by TKS

5.1. Command line arguments:

The scriptengine behaviour can be adjusted by a number of command line arguments; following the common GNU style each option may be abbreviated.

```
$ tks -h
TKS - an extensible c++ glue script language 0.5.2.4. (C) 2003 Bastian Spiegel.
          -- visit http://tkscript.de for further information.

 USAGE:
        tks source [args]
            - execute a project (.tkp) or portable archive (.tkx)

        tks source.tks [args]
            – execute a single source file

        tks source.tkx [args]
            – execute a portable archive  (under Windows you may also try e.g.
              "tks http://tkscript.de/tkx/bobfield.tkx")

        tks source.tkp [args]
            – execute an unpackaged project (usually for testing purposes)

        tks -c    --compile <proj>
            - compile a project (proj.tkp) to a portable TKX archive (proj.tkx) for distribution purposes

        tks -c    --compile proj.tkp [proj.tkx]
            - compile a project to a portable TKX archive for distribution purposes

        tks -d    --debuglevel nr
            - enable debug output, 1..99

        tks -da   --disassemble
            - print p-code for JIT compiled source sections

        tks -fi   --forceint
            - disable the JIT compiler

        tks -h    --help
            - print this help text

        tks -l    --list proj.tkx
            - list the contents of the given TKX archive

        tks -mp   --modulepath </path/to/dir/>|<drive:\path\to\dir\>
            - set the module search-path

        tks -nbc  --noboundarycheck
            - disable runtime array boundary checks

        tks -nlf  --nolocalfiles
            - disable access to local files (except the ones listed in the project file)

        tks -pp   --pluginpath </path/to/dir/>|<drive:\path\to\dir\>
            - set the search-path for native C/C++ extensions

        tks -q    --quiet
            - disable debug output (trace)

        tks -v    --version
            - display version number (e.g. 0.5.2.6)
```

6. Projects and TKX archives

TKS uses a virtual file system (VFS) which allows to transparently access packaged TKX archives as well as real local filesystems.

The VFS is read-only and all contained files have to be specified in a project file (TKP), which is used to map *logical* to *local* file names. The project file is evaluated when testing a project during development and finally when the project is converted to a compressed TKX archive.

This process makes it possible to package an application in a single, compact and gzip compressed file which can easily be distributed / published. Furthermore, exchanging e.g. an image file for another is merely a matter of changing the project file; no knowledge of the actual script-code is required.

Example:

```
[project]
name="example"
created="8.November 2003"
version="0.318309886"

[authors]
"programming"   =Bastian Spiegel <bs@tkscript.de>"

[chapter]
name="main"
"code/mymodule.tks"          ="mod:mymodule.tks"
"code/main.tks"              ="main.tks"
"data/any"                   ="data/foo.bar"
```

6.1. Accessing local files:

The possibility to explicitely access local files still exists since this is often required by applications, especially by those dealing with large media files like full motion video and the like.

Most file operation methods come in two flavours: one to operate on logic files possibly located in read-only TKX archives and another one to explicitely access real local files.

To avoid the problems that arise with the different kinds of presentations of pathnames under different operating systems, absolute pathnames should be avoided if possible.

The / char, which fortunately has become common even under Microsoft Windows (which uses the \ char by tradition) should be used to separate directories. Drive-letters and mount points should not be used for portabilities sake.

Note: For reasons of security, the access to local files (except the ones listed in the project file) can be entirely prevented by using the command line switch -nolocalfiles.

6.2. Script source files:

Scripts are placed in the (virtual) `code/` directory; all files in there are processed in the given order upon application start. The prefix `mod:` substitutes the current module searchpath.

Please consider that the script sources will be included in the packaged archives (-in order to be compiled to bytecode, then native code at runtime, if the application sources contain `compile { /*...*/ }` blocks).

6.3. Creating packaged `TKX` archives:

A `TKS` project (`TKP`) can be compiled to a compressed `TKX` file which will then contain all files required for execution, i.e. those listed in the project file.

Example:

```
tks -c bobfield.tkp
```

The suffix `.tkp` may be omitted:

Example:

```
tks -c bobfield
```

Afterwards, a `bobfield.tkx` file should be located in the current directory.

Hint: The creation of a `TKX` file requires `zlib` support.

6.4. Publishing TKX archives:

The world wide web can be used to distribute a `TKX` file to other users. The file must be uploaded to a web server which has been configured to recognize the `application/x-tkx` file type. `TKS` also uses the `MIME` (**M**ultipurpose **I**nternet **M**ail **E**xtension) type mechanism on the client side to integrate into the operating system and/or the web browser.

The following section describes the server-side configuration of the widely used `Apache` Webserver although this example can easily be carried forward to other systems:

```
pxxxxxxxx@kundenserver:~ > cat .htaccess
AddType application/x-tkx tkx
AddType application/x-tks tks
```

This will assign unique `MIME` types to files using the suffices `.tkx` and `.tks`.

By installing the `TKS` runtime module (`tks-setup.exe`) these `MIME` types are bound to the `tks` application so that it will automatically be started when a user selects a `.tkx` file; e.g. by clicking on a link in a web- or filebrowser.

Note: Users of alternative operating systems (e.g. `Linux`) need to manually take care of the `TKX MIME` type integration. This can usually be done easily using the browsers user interface (`KDE`, `GNOME`..).

7. Modules and comments

Modules are used to create re-usable script libraries.

A module provides a collection of constants, variables, functions and classes which typicially serve specific purposes like e.g. loading and displaying fonts (`pixelfont.tks`) or providing a debug facility (`debug.tks`).

In conjunction with `C/C++` extensions (*plugins*) modules can be used to define a (reference) user interface for a plugin. An example was a a scripted voice manager for an audio synthesizer which relies on native plugin functions to generate/process waveforms

7.1. Declaration of modules:

Each module is associated with a `TKS` sourcecode file. In order to access specific modules, unique names can be assigned; thats what the module statement does.

Example:

```
module MMyModule;
```

7.2. Module archives:

A module is typically used by multiple applications which suggests the use of a common directory. The path to this common directory is specified either by the environment variable `TKS_MODULE_PATH`, by a registry entry (`Win32, HKLM\Software\TKS\TKS_MODULE_PATH`) or by the command line parameter `--modulepath` / `-mp`.

Please notice the trailing "slashes":

```
tks -mp 'c:\Programme\tks\modules\'   or
tks -mp '/usr/local/tks/modules/'
```

The project directory has priority over the common module directory which means that if a copy of a module is made in order to customize it for a specific application and placed in the project directory, this copy will be used instead of the archived module.

7.3. Scope of constants and classes:

Classes and constants declared in one module are also visible in all other modules.

7.4. Scope of functions:

Functions need to be declared before their use and are only visible within the module in which they have been declared (and implemented). Functions in other modules can explicitely be accessed by prepending the module's name.

Example:

```
module Main;
MMyModule.MyFunction(); // assumes a MYModule.MyFunction function
```

7.5. The main module:

The `Main` module is the starting point of an application. If the module contains a `main()` function, this function will be called once the project and/or `TKX` archive has been loaded successfully.

Example:

```
module Main;
function main()
{
	trace "got " + Arguments.numElements + " argument(s).";
}
```

If the module statement is omitted, `Main` is assumed.

7.6. Program code outside of functions:

If the `main()` function is missing, the application will terminate after execution of all global statements. This can be used to initialize arrays or run other precalculations (e.g. loading/generating images).
The order of execution is determined, similar to the order of compilation, by the order given in the project file.

7.7. Comments:

For documentation purposes it is highly recommended to add comments..

The keyword `//` marks the beginning of a line comment, i.e. all following chars are excluded from interpretation.

Example:

```
// ---- file  : mymodule.tks
// ---- author: Bastian Spiegel
```

The keywords `/*` und `*/` are used to exclude a paragraph from interpretation. Unlike the `//` line comment, these comments may also stretch across several lines.

Nesting of comments is possible yet deprecated since it confuses the syntax highlighting modes of most texteditors (and actually makes no sense either)

Example:

```
for(int i=0; /* this is an embedded comment */ i++; i<10)
{
	/* ..../*do something useful*/... */
}
```

8. Identifiers, delimiters, keywords and constants

Basically, TKS uses the regular 7bit ASCII charset; support for UTF-8 (unicode) ist not available yet (but may be in future releases).

8.1. Escape-Sequences:

In order to include non-printable ASCII control codes (e.g. line feed) as well as special characters like ' or \ in Strings or character based integer literals, the following char sequences are supported:

Sequence	Description
\\	backslash
\'	single quotation mark
\"	double quotation mark
\n	linefeed
\r	carriage return
\t	tabulator
\v	vertical tabulator
\f	form feed
\b	backspace
\a	alert (beep/flash screen)
\e	ESC character (decimal 27, octal 033).

The latter sequence marks the beginning of ANSI control sequences. A rather comprehensive list of ANSI escape sequences can be found here:
http://astronomy.swin.edu.au/~pbourke/dataformats/ansi.html

Note: The regular windows console emulator (command.com) and thus all shells based on it (e.g. CygWin) unfortunately do not support ANSI sequences (CygWin seems to be able to emulate ANSI sequences but you need to compile a native CygWin version of the tks.exe).

8.2. Identifiers:

Identifiers are used to define unique names for variables, classes, functions and constants. Identifiers are *case-sensitive* which means that e.g. MyVariable and myvariable are clearly distinguished.

The first char of an identifier has to be a letter [a-zA-Z] or the underscore _; identifiers must not contain delimiters and operators; reserved keywords may also not be used as identifiers.
The length is practically unlimited (memory, 24bit) but it should not exceed 128 characters.

8.3. Delimiters and operators:

The following chars separate words which are interpreted individually:

```
= >     <     ==    <=    >=    !=    ,     !
&&      ||    ++    --    +     -     *     /     &
| ^     %     <<    >>    +=    -=    *=    /=
&=      |=    ^=    %=    <<=   >>=   (     )     {
} [     ]     ;
```

The meaning of these, at first glance cryptic, char sequences is going to be explained in detail but basically they follow the standards set by popular languages like e.g. `JavaScript` and/or `C`.

8.4. Reserved keywords:

The following character sequences must not be used as identifiers:

```
abs         acos          argb          asin
asm         break         case          clamp
class       compile       cos           default
#define     define        deref         dtrace
do          else          enum          exp
false       float         for           function
frac        if            int           ln
loop        module        null          Pointer
prepare     return        rgb           rnd
round       sin           String        switch
tag         tan           tcchar        tcfloat
tcobject    tcstring      this          trace
true        while         wrap          use
```

These keywords are used for

- function calls (`abs`, `acos`, `argb`, `asin`, `clamp`, `cos`, `deref`, `dtrace`, `exp`, `frac`, `ln`, `rgb`, `rnd`, `round`, `sin`, `tan`, `tcchar`, `tcfloat`, `tcobject`, `tcstring`, `this`, `trace`, `wrap`)
- control structures / flow control (`asm`, `class`, `compile`, `do`, `else`, `for`, `function`, `if`, `loop`, `module`, `prepare`, `return`, `switch`, `while`, `use`)
- system constants (`null`, `true`, `false`, `default`).

8.5. Literals:

The following forms of representation can be used to describe constant values:

```
10                : decimal integer
10.25             : (decimal) floating point number
$fedcba98         : hexadecimal integer
#fedcba98         : hexadecimal integer using the HTML colorformat
                    (#aarrggbb, a=alpha, r=red, g=green, b=blue)
0b1101001         : binary integer
'!'               : a single ASCII char
'\n'              : a single ASCII escape sequence (ex. linefeed)
"a \'string\'\n"  : a sequence of one or more ASCII characters
"\e[2J"           : an ASCII escape sequence (ex. clear screen)
String s=
"hello, \n"
"world";          : a multi-line character sequence
```

Example:

```
trace "hello, world.";

int i=0b1101001;

float f=10.25;

String s="a \'string\'\n";

String s_clrscr="\e[2J";

String s_text="text*text*text*text*"
              "text*text*text*text*"
              "text*text*text*text*"
              "text*text*text*text*"
              "text*text*text*text*"
              "text*text*text*text*"
              ;

int c='!';

int c2=s_text[9];
```

8.6. User defined constants:

In order to improve readability, it is recommended to assign an unique name to frequently used constants so that the actual value of a constant is defined at exactly one location in the source code (typically in the header).

The statements `define` (`#define`) and/or `enum` (enumeration) are provided for this purpose.

The name of a constant must commence with a letter `[a-zA-Z]` or the underscore char. The succeeding characters may be letters, underscores or numbers `[0-9]`.

Example:

```
#define NUMLOOPS 42
loop(NUMLOOPS) { /* ... */ }
```

Example:

```
#define DEG2RAD 0.00872664625997
trace "147.5 Grad im Bogenmaß entsprechen "+(147.5*DEG2RAD);
```

Example:

```
#define AUTHOR "Bastian Spiegel <bs@tkscript.de>"
trace "This software was developed by " + AUTHOR ;
```

Example:

```
enum { RED, GREEN, BLUE }; // => RED==0, GREEN==1, BLUE==2
```

Example:

```
enum { RED, GREEN=4, BLUE }; // => RED==0, GREEN==4, BLUE==5
```

Constants are always replaced by exactly one value; a programmable macro processor like in `C` is not available.

9. Datatypes and variable declarations

The basic TKS datatypes are: integer (int), floating point number (float), char sequence (String) und (Object) Pointer. A pointer (also see p.22) is a reference to the instance of a C++ resp. script-class and thus the key to complex datastructures.

For reasons of simplicity, no further distinction between signed and unsigned integers is made.

TKS requires at least a 32bit addressation scheme; the memory consumption of an integer hereby equals the one of the Pointer datatype (i.e. 4 bytes for 32bit architectures and 8 bytes for 64bit[1]).

The script language automatically converts between the datatypes String, int and float.

Example:

```
float f="1.23";
int i=3.14;
String s=42;
```

9.1. Core API datatypes:

Each of the following datatypes is represented by a C++ class; they are always available since they are built into the scriptengine.

```
Object      - base class for all API classes; cannot be instanciated
Buffer      - an Array of bytes, derived from Stream
Class       - a user defined script class
ClassArray  - an Array of user defined script classes
Envelope    - a FloatArray (time/value couples) supporting interpolation
Event       - basically a time-stamped String
File        - used to access filesystems (local or logic (archives))
Float       - object representation of a float
FloatArray  - an Array of floats
HashTable   - an associative Array
IntArray    - an Array of ints
Integer     - object representation of an int
ObjectArray - an Array of (uniform) objects
Pointer     - a reference to an ("unknown") object
PointerArray- an Array of (possibly "mixed") Objects
Pool        - an unordered ObjectArray feat. optimized memory managment
Stack       - a LIFO (Last In First Out) memory for recursive funct. calls
Stream      - base class for Files und Buffers
String      - a (buffered) char sequence (has special fastpaths)
StringArray - an Array of char sequences
Time        - describes a point in time
```

The special classes Integer and Float provide an "*object hull*" for the scalars int und float, which comes in handy when a reference to these basic datatypes has to be created.

1A 64bit release should not be expected before end of 2004 since I do not have such hardware at my disposal, yet.

9.2. Variable declarations:

```
vardecl            ::= <type> <identifierlist> ;
type               ::= int | float | String | Pointer | <Class>
identifierlist     ::= <identifier>[=<expression>] |
                       <identifierlist>,<identifier>[=<expression>]
identifier         ::= <idchar1> | <idchar1>,<idcharseq>
identifierlist     ::= <idchar> | <idcharseq><idchar>
idchar1            ::= [a-z,A-Z,_]
idchar             ::= [a-z,A-Z,_,0-9]
expression         ::= ...arbitrary expression, e.g. "(1+2)*3"...
```

Variables may be declared wherever a statement is expected.

Example:

```
int i=42;
float x,y=2.3,z; // declare multiple vars of the same type
String s,t="hello, world.",u;
```

Example:

```
int i;  // declare a global variable (as far as this is the entire
           script)
```

Example:

```
for(int i=0; i<10; i++) trace i; // see above
```

Example:

```
function MyFunction() {
     // declare a local variable, not visible outside the function
     String s="hello, world.";
     trace s;
}
```

Example:

```
MyClass::MyMethod {
     // ---- see above ----
     String s="hello, world.";
     trace s;
}
```

Scope:

Once declared, a **global** variable is visible in the current module. Global variables in other modules are accessible by prepending the module's name (e.g. `MMyModule.myvariable`).

Variables declared in functions, i.e. local variables, are only visible within the scope of the function in which they have been declared. Local variables are always **static**, i.e. their value is not reset when a function is entered thus making recursive function calls impossible, at least without the help of the `Stack` class (see p.54).

Last but not least there is the special qualifier **local** which hints the `JIT` compiler to place certain variables on the `CPU` stack instead of using absolute memory addresses.

Example:

```
compile { local int i=0; loop(100) i++; }
```

10. Object references (Pointers)

Object references are used to avoid time and memory costly copies of objects, for instance when object parameters are passed to functions or methods.

Multiple object variables may point to one and the same object but the `TKS` runtime takes care of memory management; i.e. only one pointer at a time may be the owner of an object and thus have the right to actually remove the object from memory.

Pointers do not include type information about the object they point to; this information is rather obtained from the object itself given that it is allocated and not `null`.

An assignment of a type incompatible object datatype to a typed object variable will cause the script-engine to abort resp. raise an error when trying to use the variable.

10.1. Typed pointers:

Example:

```
// Pointer assignment, t2 und t will point to the same memory area
// after this assignment. t will be the owner, t2 is deleted before
// the assignment
Time t, t2 <= t;
t2.now();
```

Example:

```
// Pointer assignment, unbound pointer returned by "new" expression
// is bound to an (untyped) variable which previous content is
// deleted before the assignment
Time t <= new Time;
t.now();
```

10.2. Untyped pointers:

The virtual datatype `Pointer` is used to store references to arbitrary objects:

Example:

```
// unbound pointer returned by "new" expression is bound to an
// untyped variable
Pointer p <= new Time;
// this pointer is now dereferenced and bound to variable t
Time t<=deref p; t.now();
// the variable t becomes the owner of the object after the pointer
// assignment (<=). Since 'p' is already unbound, the following
// statement does not affect 't':
p<=null;
```

Example:

```
String s="hello, world.";
Pointer p <= s;  // Assignment to untyped object variable
String t <= p;   // Assignment to typed object variable
trace "t="+t;    // Debug-output of string content
trace "p="+p;    // Debug-output of memory address and type
```

10.3. Deleting a pointer:

An object reference is deleted by assigning `0` (`null`) :

Example:

```
String s="hello, world.";
// Odelete object reference; the object will be removed from memory
// since 's' is the object's owner
s<=null;
trace "s="+s; // the pointer is <void> (0) now
```

10.4. Pointer arguments:

As already mentioned, object references play an important role in providing parameters to (script-) functions and methods:

Objects are always passed by reference (*call-by-reference*) with exception of the datatype `String` which is, just like the scalars `int` und `float`, passed as a copy (*call-by-value*).

Example:

```
function MyFunction(String _s) {
      // modifies a copy of the parameter value
      _s.append(", world.");
      return _s;
}
trace MyFunction("Hello");
```

By using the `Pointer` datatype the *call-by-reference* behaviour can be forced for `Strings` , too:

Example:

```
function MyFunction(Pointer _s) {
      // untyped pointer is bound to local variable and thus becomes
      // typed
      String s<=_s;
      // directly modifies the parameter value
      _s.append(", world.");
   return _s;
}
trace MyFunction("Hello");
```

Parameters to *native* (`C/C++`) API class methods or functions are always passed by reference; like for example the parameter to the `append()` method of the `String` class or the the one for the `trace` (`C`) function in the above examples.

11. Strings

A String is composed of an arbitrary (limited by memory) number of bytes which are encoded as described by the ASCII standard. The special value 0 denotes the end of a char sequence (ASCIIZ). Currently, there is no support for UNICODE strings but this may change in future versions by using the UTF-8 encoding.

TKS strings are objects, which, besides the actual char data, also store the number of chars used (length property, includes terminating 0), the maximum number of chars available (bufferLength property) as well as an (internal) ownership flag used for constant strings.
By this means it is possible to reserve memory for successive append operations to avoid unnecessary string copies.

Example:

```
String sbuf;
sbuf.alloc(1024);
sbuf.empty();   // reset the number of used chars
sbuf="hello, ";
sbuf.append("world."); // no buffer resizing necessary
```

The buffer of a String object may also point to a constant, invariant char sequence; an alterative operation on such a String will automatically create a copy first.

Example:

```
String s<="hello";  // set buffer to a constant string
// the following append operation will automatically create a copy
// which will be deleted after the statement has been executed.
trace s+", world";
```

A comprehensive description of all supported string operations is available in the API reference; only a listing is given here:

```
operator !=(), operator &(), operator &&(), operator >(), operator >=(),
operator <(), operator <<(), operator <=(), operator +(), operator ==(),
operator [](), operator ||(), alloc(), append(), copy(), empty(),
endsWith(), fixLength(), free(), freeStack(), getBufferLength(), getc(),
getLength(), getWord(), indexOf(), insert(), isBlank(), lastIndexOf(),
load(), loadLocal(), parseXML(), patternMatch(), print(), putc(),
replace(), saveLocal(), split(), startsWith(), substring(), toLower(),
toUpper(), trim(), words()
```

The [] operator is used to access single chars of a String. In doing so, one needs to be aware that it is not allowed to access chars beyond the ASCIIZ character (i.e. 0..index<string.length).

Example:

```
String s<="hello, world.";
trace "the 6th char of the string is:\'"+tcchar(s[5])+"\'.";
```

The function tcchar() is used to convert an ASCII character code (s[5]=44==',') into a printable (2 char long, including ASCIIZ) String (",") .

11.1. Stringlists

Since working with strings often requires to split them into substrings, for instance when reading text files whose lines are typically separated by the `\n` (*newline*) character, the `String` datatype features a builtin list mechanism. The methods `split()` and `words()` are used to create these string lists.

Example:

```
// load a local text file (true=ASCII mode, carriage returns ('\r')
// are deleted) and split it into lines
String t,s; s.loadLocal("test.txt", true); s.split('\n');
foreach t in s trace "t="+t;
s.freeStack();
```

Example:

```
// split into words, do not take embedded strings into account:
String t,s<="abc def ghi jkl \". . .\""; s.words(false);
foreach t in s trace "t="+t;
s.freeStack();
```

Example:

```
// split into words, take embedded strings into account, these
// are going to be interpreted as one word:
String t,s<="abc def ghi jkl \". . .\""; s.words(true);
foreach t in s trace "t="+t;
s.freeStack();
```

Example:

```
String s<="one two three"; s.words(1);
trace "the 2. word is "+s.getWord(1);
s.freeStack();
```

The list of a `String` object has to be freed manually after usage. Successive calls to `split()` or. `words()` will not have the desired effect otherwise.

11.2. Formatted text files

Text files are often stored in a machine-readable form in order to allow for automatic processing. XML, the Extensible Markup Language, has evolved into a standard format for these purposes.

XML defines the basic structure and syntax of a text file. Each XML file typically starts with a reference to the Document Type Description (DTD), which an XML parser should use to validate the structural configuration of a document. The DTD describes the set of elements, their attributes and possible values and how the elements may be nested to form complex data structures.

TKS uses a simplified XML parser which only performs a basic syntax check (e.g. <e> must be closed with </e> on the same level). Flow text between elements and DTD validation are not supported.

The String.parseXML() method splits a String into an L/R tree; the *left* nodes link elements of the same hierarchy level, the *right* linked nodes lead to subtrees of an element structure. The attribut lists of elements are converted to HashTables (associative arrays). The element structure of the document is converted to TreeNodes, which store the element HashTables, to make it accessible from scripts.

Example:

```
String s<="<test><body><text value=\" \'. . .\' <test>\"/></body></test>";
TreeNode t<=s.parseXML();
TreeNode u<=t.right; trace "u.name="+u.name; u<=u.right;
HashTable r<=u.object; trace "text=\""+r["value"]+"\"";
```

12. Operators

Operators play an import role in assign-statements and expressions; complex expressions are created by combining two or more expressions using operators.

The following operators are supported:

12.1. Operator classes and priorities:

1. Arithmetic and bit operators (+ - * / % | ^ & << >> ~)
2. Boolean operators (! || && ^^)
3. Relational operators (== <= != >= < >)
4. Post- and pre-increment/decrement operators (++ --)
5. Assignment operators (= *= /= %= &= += -= |= ^= >>= <<=)

```
High priority      : ! ~ -(unary negation)
Medium priority    : * / % &
Low priority       : + - | ^ || && ^^ == <= != >= < > >> <<
```

Example:

```
// first evaluate the expression 3*2, then subtract the result (6)
// from 20, then add 10. Thus, the result is 24.
int i = 10 + 20 - 3 * 2;
```

In order to create self contained subexpressions, for instance to assign priorities to them, an expression may be enclosed in () brackets:

Example:

```
// first evaluate the expr (20-3), then multiply the result by 2 and
// finally add 10. Thus the result is 44.
int i = 10 + (20-3) * 2;
```

12.2. Assignment operators:

The assignment operators are used to store the result value of an expression in a memory location, e.g. global and local variables, array and hashtable elements and class members.

If the = char is preceded by an arithmetic operator, the previous value of a memory location can be taken into account when calculating its new seizure.
Consequently, the assignment operators are shortforms for the following assignment operations:

```
l  *= <expression>;      equals    l = l  *  <expression>;
l  /= <expression>;      equals    l = l  /  <expression>;
l  %= <expression>;      equals    l = l  %  <expression>;
l  &= <expression>;      equals    l = l  &  <expression>;
l  += <expression>;      equals    l = l  +  <expression>;
l  -= <expression>;      equals    l = l  -  <expression>;
l  |= <expression>;      equals    l = l  |  <expression>;
l  ^= <expression>;      equals    l = l  ^  <expression>;
l >>= <expression>;      equals    l = l  >> <expression>;
l <<= <expression>;      equals    l = l  << <expression>;
```

Caveat:

The target of an assignment must not be a complex expression, `a.b.c.d=e;` is not allowed for instance. The following assignment targets are currently possible:

```
myvar <assignop> <expr>;
MyModule.myvar <assignop> <expr>;
myclassinstance.member <assignop> <expr>;
myarrayvar[<idx>] <assignop> <expr>;
myhashvar[<idx>] <assignop> <expr>;
myapiclassinstance.mymember <assignop> <expr>;
myclassinstance.mymember <assignop> <expr>;
myclassinstance.myarrayvar[<idx>] <assignop> <expr>;
myclassinstance.myhashvar[<idx>] <assignop> <expr>;
```

The order of evaluation of subexpressions on the same priority level is not defined and rather up to the runtime module being used (e.g. `JIT` runtime vs. interpreted mode).

Short-Circuit evaluation is not supported, i.e. all subexpressions are evaluated regardless whether the result of the entire expression could be determined already by its first subexpression:

Example:

```
if(i && j && k) { /* ... */ }
```

This statement can be optimized like this:

Example:

```
if i if j if k  { /* ... */ }
```

12.3. Unary operators:

Unary operators have the highest evaluation priority. In contrary to `C++` or `Java` there is no unary typecast operator; this functionality is rather provided by builtin function calls (`tcint()`, `tcfloat()`, `tcchar()`, `tcstring()`).

Example:

```
int i = -1;  // negation, result is -1.
```

Example:

```
int i = ~1;  // bitwise not, result is -2
             // (0xFFFFFFFE on 32bit architectures)
```

Example:

```
int i = !1;  // logical not, result is 0
```

12.4. The ternary operator:

This operator is used to choose between two sub-expressions depending on the result of a conditional expression. Please see p. 31 for details.

Example:

```
String s = rnd(2) ? ”true” : ”false”;
```

12.5. The streaming << operator:

The << operator is used to (de-)serialize objects from and to I/O stream objects.

Example:

```
String s="hello, world.";

Buffer b; b.size=512; // instanciate a Buffer(Stream)

b << s; // serialize the string into the Buffer(Stream)

s="..."; trace s; // unset string

b.offset=0; // reset stream offset

s << b;  // deserialize the string from the Buffer(Stream)

trace "s="+s;
```

Example:

please see http://tkscript.de/4s/testfileio.tks.html .

13. Expressions

Expressions are used to combine one, two or three values (subexpressions) to a result value by using operators. The most simple example for an expression is a constant value.

Example:
```
int i=42;  // constant expression ("42")
int j=i;   // variable expression ("i")
```

From a technical point of view, the result of every expression is exactly one value. Since this value may represent a reference to an array or other complex datastructure, an expression may also evaluate to an array of values *effectively*.

Example:
```
// the result of the expression is a pointer to an IntArray which
// again is made up by an arbitrary number of values
function ReturnN {
     return [1,2,3,4,5,6,7,8];
}
```

A variable expression may also contain a pre/post increment/decrement operator (++v, v++, --v, v--) which is used to alter the value of a variable without explicite assignments. The variable is incremented resp. decremented, depending on the position of the operator, before or after it is saved for later evaluation. This works with int as well as float variables (+1.0, -1.0).

Example:
```
int i=42, j=i++;  // j is 42, i becomes 43.
```

Example:
```
int i=42, j=--i;  // j is 41, i becomes 41.
```

This may also be applied to members of modules and classes:

Example:
```
int j = MMyModule.myvariable++;
```

Example:
```
int j = --myobj.mymember;
```

These operators may also be used in statements:

Example:
```
int i=42; i++; i--;
```

Boolean expressions:

A boolean expression evaluates to either `true` (`!=0`) or `false` (`==0`). Boolean expressions are typically used to test break conditions of loops or to signalize conditions using flags (or shifted bitmasks). The integer literals `true` and `false` can be used to substitute the values `1` and `0`.

13.1. the range expression:

Example:

```
int i=5; i= (1 < i < 10);
float f=5.23; i= (1.1 < f < 10.10);
```

The range expression tests whether an int or float value ranges between the given boundaries.

13.2. The “mini-if” ternary expression:

This expression tests a condition within an expression and selects between two alternative result expressions depending on the test result, which may be `true` or `false`.

Example:

```
// the value of the ternary expression is determined by
// the seizure of the variable i
int i=rnd(10),j=rnd(10),k=i>5?(i+j):(j-i);
```

The above example equals the following program in terms of functionality:

```
int i=rnd(10), j=rnd(10), k;
if(j>5)
      k=i+j;
else
      k=j-i;
```

13.3. The `new` expression:

The new expression is used to explicitely create new objects.
The argument passed to this expression must be the name of a user-defined script- or a`C++` API class. This expression is rarely needed in `TKS` but it is surely important for o*bject factories*.

Example:

```
function NewString {
      String s<=new String;  // allocate new String
      return deref s; // release binding to variable s
}
String mynewstring<=NewString();
```

Example:

```
class AbstractFactory {
      newObject() {/*trace "Abstract::newObject, never called";*/}
}
class ConcreteStringFactory : AbstractFactory {
      newObject() { return new String; }
}
class ConcreteFloatFactory : AbstractFactory {
      newObject() { return new Float; }
}
```

14. Statements

Statements are used to declare variables, constants, classes, functions and modules. A statement has no return value but is rather used to either store the result value from its contained expression(s) to a memory location or use it as a parameter for a control structure. TKS supports the following control structures:

- `do..while` (p.34)
- `if..else` (p.33)
- `for` (p.36)
- `foreach` (p.38)
- `loop` (p.37)
- `switch..case` (p.35)
- `while` (p.34)

other supported statements are:

- `clamp` (p.39)
- `class` (p.45)
- `define` (p.16)
- `enum` (p.16)
- `function` (p.42)
- `module` (p.14)
- `prepare` (p.39)
- `use` (p.40)
- `wrap` (p.39)
- *<builtin function calls>* (p.44)
- *<class method calls>* (p.45)
- *<variable/array/hash/member assignments>* (p.27)
- *<variable declarations>* (p.20)
- *<user defined function calls>* (p.42)

Statements are often grouped in blocks (*statement sequences*) which are enclosed in {} brackets.

For every source file, a top level statement sequence is implicitely created which makes it possible to write a TKS script which just consists of a single statement, for instance.

Statements outside of functions and methods are executed after successful compilation of all modules defined in the project file. The order of execution hereby equals the order of occurrence in the project file. This mechanism can be used to initialize objects, allocate arrays and/or initialize variables.

Example:

```
trace "hello, world.";
```

...represents a complete, valid TKS program.

In contrary to C, a statement block may not directly contain another statement block:

Example:

```
if(i==42) { { /* ..this does not work.. */ } }
```

Example:

```
if(i==42) { if(j==1) { /* ..this is allowed.. */ } }
```

14.1. The `if..else` statement

This statement is used to branch conditionally depending on the result of a boolean expression.

If the expression evaluates to `true` (`!=0`) then the statement after `if` will be executed, otherwise the control flow will branch to the `else` statement, if available.

Example:

```
if(i==42) j=1;
```

Example:

```
if(i==42) j=1; else j=2;
```

Example:

```
if(i==42) { j=1; k=2; } else { j=2; k=3; }
```

Note:

The `()` brackets are not required; actually they are part of the expression:

Example:

```
if i==42 j=1;
```

Example:

```
if i==42 j=1; else j=2;
```

Example:

```
if i==42 { j=1; k=2; } else { j=2; k=3; }
```

14.2. The `do..while` statement

This statement is used to loop a given statement (-sequence) until the boolean expression after `while` evaluates to `false` (`0`).

Example:

```
int i=0; do i++; while (i<10);
```

Example:

```
int i=0,j=0; do { i++; j++; } while (i+j)<10;
```

The boolean condition will be tested each time a loop iteration finishes, i.e. the `do..while` loop body is run at least once.

Similar to the `if` statement, the `()` brackets around the conditional expression are optional and actually part of the expression.

14.3. The `while` statement

This statement is used to repeat a statement sequence as long as the boolean expression after `while` evaluates to `true` (`!=0`).

Example:

```
int i=0; while(i<10) i++;
```

Example:

```
int i=0,j=0; while(i+j)<10 { i++; j++; }
```

14.4. The `switch` statement

This statement is used to test the value of an expression for a number of conditions; if a condition is met, the associated statement sequence will be executed. The condition `default` hereby refers to all conditions not listed explicitely.

Each statement sequence has to be closed using the `break` keyword since otherwise the sequence of the next condition will be executed as well (*run into case label*) regardless whether that condition is met or not.

Example:

```
int i=rnd(10);
switch(i) {
case 1:
      trace "i is 1.";
      break;
case 2:
      trace "i is 2.";
      break;
case 3:
case 4:
      trace "i is 3 or 4";
      break;
default:
      trace "i is neither 1,2,3 nor 4.";
      break;
}
```

In contrary to `C`, also `String` and `float` condition expressions are allowed; furthermore, case expressions do not need to be constant but may contain even function calls.

Example:

```
function MyFunction() {
      return 10;
}

String s=Arguments[0];

switch(s) {
   case "-h":
   case "--help":
      /* ... */
      break;
   case MyFunction(): // case 10
      /* ... */
      break;
}
```

14.5. Die `for` Anweisung

This statement is used to repeat the loop body as long as the loop condition, a boolean expression, evaluates to `true`. Thus, the mode of operation of the `for` loop is similar to the `while` loop, although additional initialization and modification statements may be provided. The empty statement (`;`) is used to omit the initialization; no semicolon is required to terminate the modification statement.

The initialization statement is called before the first loop iteration starts, the modification statement is called every time an interation finishes. Before each iteration, the loop condition is tested and if it evaluates to `true`, the loop body is executed and the next iteration starts.

Example:

```
for(int i=0; i<10; i++) trace "i="+i;
```

```
trace "i="+i;     is the loop body
     int i=0;     is the loop initialization
        i<10;     is the loop condition
         i++      is the modification statement
           i      is the loop counter
```

Example:

```
int i=0;
for(;i<10;) trace "i="+i++; // omit the init + modif. statements
```

Example:

```
int n = 16; // http://www.bagley.org/~doug/shootout/nestedloop
int x = 0;
for (int a=0; a<n; a++)
    for (int b=0; b<n; b++)
        for (int c=0; c<n; c++)
            for (int d=0; d<n; d++)
                for (int e=0; e<n; e++)
                    for (int f=0; f<n; f++)
                        x++;
```

14.6. The `loop` statement

This statement resembles the `while` and `for` statements. Actually, it represents a frequently used special case of the `for` resp. `while` statement.

The expression given after the `loop` keyword determines the number of iterations. This integer expression is evaluated only ***once*** before the first iteration starts. There is no way to access the internal loop counter, it will simply be decremented until it reaches `0`.

The brackets around the integer expression are, similar to the `do..while`, `while` and `for` statements, technically part of the expression.

Example:

```
loop 10 trace "loop.";
```

Example:

```
int i=0;
loop(10) trace "i="+i++;
```

Example:

```
int n = 16; // http://www.bagley.org/~doug/shootout/nestedloop
int x = 0;
loop(n)
    loop(n)
        loop(n)
            loop(n)
                loop(n)
                    loop(n)
                        x++;
```

14.7. The `foreach` statement

This statement is used to iterate container objects like for instance `String` lists, `Arrays`, `HashTables` or `Pools`. The loop body will be executed for each element in the given container object.

To break out of a `foreach` loop, the value `-1` can be assigned if the loop variable is an integer. If the loop variable is a pointer, the value `0` (`null`) signalizes the scriptengine to terminate the loop.

Example:

```
String t,s="one \"two and a half\" three"; s.words(true);
foreach t in s trace "t="+t;
```

Example:

```
int i; foreach i in [1,2,4,7,9,11] trace "i="+i;
```

Example:

```
int i=0; // http://www.bagley.org/~doug/shootout/bench/wordfreq/
String s,k;

HashTable words; words.alloc(20000);
s.loadLocal("wordfreq.txt", true);
s.words(false);
foreach k in s {
      k.toLower();
      words[k]=tcint(words[k])+1;
}
s.freeStack();

StringArray sta; sta.alloc(words.numElements);
IntArray ita;    ita.alloc(words.numElements);
i=0;
foreach k in words {
      ita[i]=i;
      Integer io; io.value=words[k]; sta[i]=io.printf("%7d")+" "+k;
      i++;
}

sta.sortByValue(ita, false);
i=words.numElements;
loop(--i)
    trace sta[ita[i--]];

trace "words.numElements="+words.numElements;
```

14.8. The `prepare` statement

This statement is used to run one-time initializations in the context of a function. The statement sequence given after the `prepare` keyword will be executed only once during the application run.

Example:

```
function FLookUp(int _i) {
      IntArray lut;
      prepare {
            lut.alloc(512);
            int i=0;
            loop(256) lut.add(i++);
            loop(256) lut.add(255);
      }
      return lut[_i];
}
trace FLookUp(184); trace FLookUp(384);
```

14.9. The `clamp` and `wrap` statements

This statement is used to limit the value of a variable (`int`, `float` or `Object` reference) to a given range.

The clamp statement is implemented using the following `C`-code:

```
if(var->pointer.float_val<min.value.float_val)
      var->pointer.float_val=min.value.float_val;
else
      if(var->pointer.float_val>max.value.float_val)
            var->pointer.float_val=max.value.float_val;
```

The wrap statement is implemented using the following `C`-code:

```
if(var->pointer.float_val<min.value.float_val)
      var->pointer.float_val = (var->pointer.float_val
                                - min.value.float_val)
                               + max.value.float_val;
else if(var->pointer.float_val>=max.value.float_val)
        var->pointer.float_val = min.value.float_val
                                 + (var->pointer.float_val
                                    - max.value.float_val);
```

Example:

```
float f=32; clamp f 0 20; // f is 20
```

Example:

```
int i=32; wrap i 0 20; // i is 12
```

In case the given variable is of type `Object`, the limits need to have the same datatype as the variable. Hence, this mechanism can be used to validate the range of compound datatypes imported using the `C++` plugin interface.

Example:

```
use tksdl;
Vector v<=vector(2,3,4);
wrap v vector(0,0,0) vector(1,2,3); // v is (1,1,1)
```

14.10. The `use` statement

This statement is used to dynamically load native program components ("*plugins*") at runtime. Each plugin is associated with one shared object file (`.dll` or `.so` file extension). Plugins will first be searched in the current directory, then in the directory determined by the environment variable `TKS_PLUGIN_PATH` and finally in the directory stored in the registry key `HKLM\Software\TKS\TKS_PLUGIN_PATH` (Microsoft Windows only).

Example:

```
use tksdl; // request the "tksdl.so" resp. "tksdl.dll" plugin
Viewport.openWindow(640,480);
SDL.eventLoop();
```

Another use of the statement is to bind scriptfunctions to events raised by `C++` objects, please see p.77 for details.

Example:

```
function onKeyboard(Key _k) {
      trace "key \""+_k.name+"\" was pressed";
}
use tksdl;
Viewport.openWindow(640,480);
use onKeyboard for SDL.onKeyboard;
SDL.eventLoop();
```

Example:

```
function onKeyboard(Key _k) {
   trace "key \""+_k.name+"\" was pressed";
}
use tksdl;
Viewport.openWindow(640,480);
use callbacks; // auto-bind all available callbacks
SDL.eventLoop();
```

14.11. The `define` statement

This statement is used to define constant values. It is parsed while scanning the source code. A look up table is generated that is used while parsing the source code after it passed the scanner. Please see p.19 for details and examples.

14.12. The `enum` statement

Please see p.19 for details and examples.

14.13. The `function` statement

This statement is used to declare new script functions. Please see p.42 for further information.

14.14. The `class` statement

This statement is used to declare new script classes. Please see p.45 for further information.

15. Functions

Functions are used to group frequently used statement sequences in order to avoid redundancy. These statement blocks may be parametrized by up to 255 resp. 8 arguments. Thus, functions support modularization of programs and furthermore help to split a problem into (possibly independent) sub problems (*Divide and Conquer*).

Basically, a distinction between user defined script functions and those being imported through the `C/C++` plugin interface must be made. Plugin functions are limited to 8 arguments and `Strings` are always passed by reference while scripted functions may take up to 255 arguments and use the *call-by-value* strategy for `Strings`.

15.1. User defined functions

The return value of a user defined function is dynamic and may change according to the current parameter set. Each function may return one value at most. If multiple return values are required, a `HashTable` or `Array` object can be used.

Example:

```
function Deg2Rad(float _degrees) {
	return _degrees*2PI/360.0;
}

trace Deg2Rad(90);
```

Example:

```
function SplitFloat(float _f) {
	float ffrac=frac(_f);
	return [_f-ffrac, ffrac];
}

float f;
foreach f in SplitFloat(1.23)
	trace f;
```

15.1.1. Forward declarations:

In contrary to classes, functions need to be declared before their first use. In case the implementation does not immediately follow the declaration, the declaration will be called *Forward Declaration.* The argument list may not be repeated when a forward declarated function is implemented.

Example:

```
function HTMLSpan(String _s); // "Forward Declaration"
 HTMLSpan("test");
 function HTMLSpan {
	// Implementation, argument list is not repeated
	return "<span class=myfun>"+_s+"</span>";
}
```

As already mentioned on p.23, `String` arguments to user defined script functions are passed by value.
In the above example, an unnecessary copy of the parameter to the function `HTMLSpan()` is created since it will only be read, not modified.

Consequently, the function can be optimized in the following way:

Example:

```
function HTMLSpan(Pointer _s);
HTMLSpan("test");
function HTMLSpan { return "<span class=myfun>"+_s+"</span>"; }
```

15.1.2. Functions in other modules

Functions in other modules can be accessed by prepending the respective module name:

Example:

```
MMyModule.MyFunction();
```

15.1.3. Variable return types

The return type of a function is not fixed but may depend on the current arguments.

Example:

```
HashTable values<=#["i"=42, "f"=3.14, "s"="hello, world."];

function vreturn(String _arg) {
     if(values.exists(_arg))
          return values[_arg];
     else
          return "err: no such element "+_arg+".";
}
```

15.2. TKS API functions

The following functions are provided by the `TKS` core `API`:

`int`	**`2n`**	`(int)` – *return next in size power of 2*
`int|float`	**`abs`**	`(int|float)` – *return absolute (positive) value*
`float`	**`acos`**	`(float)` – *return arcus cosine*
`int`	**`argb`**	`(int, int, int, int)` – *return packed 32Bit color value*
`float`	**`asin`**	`(float)` – *return arcus sine*
`float`	**`cos`**	`(int|float)` – *return cosine*
`float`	**`deg`**	`(float)` – *convert radian to degree*
	`die`	`(String)` – *output err. Message and abort the application*
	`exit`	`(int)` – *exit application and return errorcode to the invoking shell*
`float`	**`frac`**	`(float)` – *return decimal place*
`String`	**`getenv`**	`(String)` – *return value for environment variable*
`float`	**`mathAbsMaxf`**	`(float, float)` – *return the bigger of two absolute values*
`int`	**`mathAbsMaxi`**	`(int, int)` – *return the bigger of two absolute values*
`float`	**`mathAbsMinf`**	`(float, float)` – *return the smaller of two absolute values*
`int`	**`mathAbsMini`**	`(int, int)` – *return the smaller of two absolute values*
`float`	**`mathMaxf`**	`(float, float)` – *return the bigger of two values*
`int`	**`mathMaxi`**	`(int, int)` – *return the bigger of two values*
`float`	**`mathMinf`**	`(float, float)` – *return the smaller of two values*
`int`	**`mathMini`**	`(int, int)` – *return the smaller of two values*
`float`	**`mathPowerf`**	`(float, float)` – *return a raised to the power of b*
`int`	**`mathPoweri`**	`(int, int)` – *return a raised to the power of b*
	`putenv`	`(String, String)` – *set an environment variable*
`float`	**`rad`**	`(float)` – *convert degree to radian*
`int`	**`rgb`**	`(int, int, int)` – *return packed 32Bit color value (alpha=255)*
`int|float`	**`rnd`**	`(int|float)` – *return random number in the given range*
`float`	**`round`**	`(float)` – *round up to next integer*
`float`	**`sin`**	`(int|float)` – *return sine*
`float`	**`sqrt`**	`(int|float)` – *return square root*
	`stderr`	`(String)` – *output String to STDERR*
	`stdout`	`(String)` – *output String to STDOUT*
`int`	**`system`**	`(String)` – *execute system command**
`float`	**`tan`**	`(int|float)` – *return tangent*
`String`	**`tcchar`**	`(int)` – *convert integer to 1-char ASCII String*
`float`	**`tcfloat`**	`(int|float|String)` – *convert value to floating point representation*
`int`	**`tcint`**	`(int|float|String)` – *convert value to integer representation*
`Object`	**`tcobject`**	`(int|float|String)` – *convert value to Object representation*
`String`	**`tcstring`**	`(int|float|String)` – *convert value to String representation*
	`trace`	`(String)` – *output String to debug console (default is STDERR)*

For further information please see p.81 and `http://tkscript.de/api/index.html`.

Note: For speeds sake, most of the frequently used functions have designated codepaths.

Note: For reasons of security, the `system()` call only works if the `DX_SYSEXEC` constant is defined at compile time. Otherwise -1 will be returned.

16. Complex datastructures

16.1. Classes

A class is a structure which is composed of the scalars `int`, `float` and objects, i.e. instances of `C++` resp. user defined script classes. The elements of a class are called *members* or *properties*.

Classes are used to extend applications by new datatypes. Similar to basic datatypes like e.g. `int` or `float`, classes first need to be instanciated before they can actually be used; such instances are called `Objects`.

16.1.1. Members

A member is a variable defined in the namespace of a class. Each time a class is instanciated, memory for its members is allocated, too. Members are visible in class methods but they can also be accessed from outside script code using the "`objname.member`" syntax. Member declarations may not contain initializers (e.g. `int i=42;` or `int ia[42];`)

Example:

```
class MyClass {
      int i;
      float f;
      String s;
      int ia[]; // same as IntArray ia;
      FloatArray fa;
      HashTable ht;
}
MyClass obj;

trace "obj.f="+obj.f;
```

16.1.2. Script vs. API classes

While script classes are declared by user defined scripts, `C++ API` classes are either built into the script-engine (core `API` classes) or imported through the `YAC` plugin interface.

The following `C++` classes are part of the core `TKS API`:

`Buffer`, `ClassArray`, `Configuration`, `Envelope`, `Event`, `File`, `Float`, `FloatArray`, `HashTable`, `IntArray`, `Integer`, `ObjectArray`, `PointerArray`, `Pool`, `Stack`, `Stream`, `String`, `StringArray`, `StringIterator`, `Time`, `TKS`, `TreeNode`.

Please see p.81 for a comprehensive description of all core classes.

16.1.3. Methods

Methods are used to bundle frequently used statements which operate on objects. A method is similar to a function except that it misses the keyword "`function`" and is declared *within* a class.

Example:

```
class MyClass {
      myMethod() { trace "myMethod called."; }
}
MyClass c;
c.myMethod();
```

The method may be implemented directly within the class definition (like in the above example) or a forward declaration can be used so the method body can be implemented later on (recommended for larger statement blocks):

Example:

```
class MyClass {
      int i;
      float f;

      myMethod(int _i, float _f); // forward declaration
}

MyClass::myMethod { // implementation
      i=_i;
      f=_f;
      trace "myMethod called. i="+i+" f="+f;
}

MyClass c;
c.myMethod();
```

Methods in super classes (i.e. base classes) can be called by prepending the base class name:

Example:

```
class BaseClass {
      sayHello() { stdout "hi!\n"; }
}

class MyClass : BaseClass {
      myMethod() { stdout "myMethod says "; BaseClass::sayHello(); }
}

MyClass c;
c.myMethod();
```

16.1.4. Constructors and Destructors

In order to initialize class elements upon object instanciation, a class may provide suitable **constructors** and **destructors**.

A constructor is a method which carries the same name as the class; this also applies to the destructor with the difference that its name is preceeded by the ~ char.

In `TKS`, no parameters may be passed to constructors and destructors; appriopriate `init()` resp. `exit()` methods should be used instead.

Constructors are called when an object is created. If the class was derived from a base class, the constructor of the base class is called first.

When an object is deleted, its destructors are called in the reversed order, i.e. the base class destructor is called last.

Example:

```
class Telephone {
          float fVolume; // a member (property)

          Telephone() { // constructor
               trace "constructing Telephone.";
               fVolume=1.0;
          }

          ~Telephone(); // forward declaration of the destructor

          ring() { trace "Telephone::ring"; } // a regular method

          // define a user interface to set the volume
       setVolume(float _f) { fVolume=_f; }
}

Telephone::~Telephone { // implement the destructor
   trace "deleting Telephone.";
}

// instanciate a Telephone object, implicite constructor call
Telephone t;

t.fVolume=0.74; // directly access a property (member)

// set a property using the designated user interface method
t.setVolume(0.74);

t.ring(); // call a method

t<=null; // explicitely delete the object, implicite destructor call
```

16.1.5. Inheritance and late binding

For purposes of refinement, specialization or extension, a class may be derived from one or more base classes. A derived class inherits all members and methods of its base class(es). TKS supports multiple inheritance, i.e. each class may have up to 64 base classes.

Methods of the base class(es) may be overwritten with new implementations as long as their parameter signature (i.e. number and type of parameters) is preserved.

When an overwritten method is called, the late binding mechanism first looks up the actual method to be called. The selected method depends on the type of the object, not the type of the object variable since objects that have one or more base classes may be bound (*downcasted*) to any base class pointer variable. This process is also known as *virtual call* in C++.

Example:

```
class BaseClass {
      exec() { trace "BaseClass::exec"; }
}

class ExtClass : BaseClass {
      exec() { trace "ExtClass::exec"; }
}

BaseClass bc <= new ExtClass; // downcast to base class

// ---- late binding mechanism selects ExtClass::exec method ----
bc.exec();
```

Example:

```
class C1 {
      int i;
}

class C2 {
      float f;
}

class C3 : C1, C2 {
      C3() {
                  i=42; f=10;
                  trace "C3::C3()";
            }
}

C3 c;
```

Example:

```
class CClass {    // "interface" definition..
      outputHTML() { return "*ill*"; }
}

class CLink : CClass {
      String target;
      String label;
      getLabel() { return label; }
      outputHTML() {
            return "<a href=\""+target+"\">"+getLabel()+"</a>";
      }
}

class CImage : CClass {
      String img_source;
      String img_alt;
      outputHTML() {
            return "<img alt=\""+img_alt+
                  "\" src=\""+img_source+"\">";
      }
}

class CImageLink : CImage, CLink {
      getLabel() { return CImage::outputHTML(); }
      outputHTML() { return CLink::outputHTML(); }
}

CImageLink il;
il.target="http://tkscript.de";
il.img_source="images/test.png";
il.img_alt="test";
trace il.outputHTML();
```

16.2. Arrays and Arraylists

An array is a sequential field of `int`, `float` or `Object` elements.

From a technical point of view, an array is also an instance of a `C++ API` class, i.e. there are dedicated array classes for different element types:

```
int         ->    IntArray
float       ->    FloatArray
String      ->    StringArray
Pointer     ->    PointerArray, ObjectArray
<uclass>    ->    ClassArray
```

Each of these array classes supports the `[]` operator which is used to read or modify elements.

Example:

```
int ia[10];   // create IntArray with 10 elements
ia[2]=3;      // set an array element
trace ia[2]; // print an array element
```

The `alloc()`, `realloc()` and `free()` methods are used to determine or change the size of an array resp. free its elements.

If an array is resized using the `realloc()` method, its old content is preserved if possible.
If the new array size is smaller than the number of previously used elements, (`oldNumElements-newMaxElements`) will be discarded.

Example:

```
IntArray ia; ia.alloc(10); // create IntArray with 10 elements
ia[5]=42;       // set the 6. array element, also changes numElements
ia.realloc(20); // resize array
trace ia[5]; // read an element
```

Each array keeps track of the number of elements actually in use (`numElements`) as well as the maximum number of elements available (`maxElements`). Upon creation of an array, the `maxElements` property is initialized with the value passed to the `alloc()` method, `numElements` will be set to 0, though.

The `empty()` method is used to reset `numElements` without actually discarding the elements.

Single elements can be added resp. removed by using the `add()`, `insert()` and `delete()` methods. This mode of operation resembles lists, hence the term *array list*. In contrary to real lists, the maximum list length is limited by the number of allocated elements, though.

Example:

```
float fa[100]; loop(fa.maxElements) fa.add( rnd(1.0) );
fa.realloc(200);
loop(fa.MaxElements-fa.numElements) fa.add( rnd(1.0) );
trace "fa.numElements=" + fa.numElements;
fa.empty();
trace "empty()";
trace "fa.numElements =" + fa.numElements
    + "fa.maxElements =" + fa.maxElements;
```

16.3. Associative Arrays (Hashtables)

Unlike regular arrays, `Hashtables` are not indexed sequentially. Instead, named keys (`Strings`) are used to calculate the actual (internal) index using a so called *hash function*.

Furthermore, each `HashTable` element may have its individual type which is determined by the expression result being assigned to it. If this result value represents an unbound pointer, no explicite pointer assignment is required to bind that pointer to a `HashTable` element slot.

Example:

```
HashTable ht; ht.alloc(113);
ht["myint"]=42;
ht["myfloat"]=1.23;
ht["mystr"]="hello, world."; // bind constant pointer
ht["myobj"]=new IntArray; // bind r/w pointer to HashTable slot
if ht.exists("myobj")
     trace "found slot \"myobj\"";

ht.delete("myobj");
```

Example:

```
int n=150; // http://www.bagley.org/~doug/shootout/bench/hash2
HashTable hash1; hash1.alloc(10000);
HashTable hash2; hash2.alloc(10000);
int i=0;
for(i=0; i<10000; i++)
    hash1["foo_" + i] = i;
String k;
loop(n)
    foreach k in hash1
        hash2[k] += hash1[k];
```

16.4. Array initializers

The values of array elements may be set by either assigning single values or by using the following array initializer expressions:

Example:

```
StringArray str <= [ "one", "two", "three"];
FloatArray  flt <= [ 1.1, 2.2, 3.3 ];
IntArray    ia  <= [1, 2, 3];
HashTable   ht  <= #["myint"=42, "myfloat"=1.23,
                     "mystr"="hello, world.", "myobj"=new IntArray];
```

The first element of an array initializer determines the type of array to be created. `HashTable` initializers are preceeded by the `#` char.

Note:

The return value (i.e. an array object) of an array initializer is constant, i.e. read-only! Nevertheless, when the initialization expression is called more than once, the expression lists are evaluated again and the elements of the array are updated.

16.5. Multidimensional arrays

In TKS, a multidimensional array is constructed using an array of pointers that point to IntArrays, FloatArrays, StringArrays or PointerArrays:

Example:

```
int i;
PointerArray mda <= [ [1,2,3], [4,5,6], [7,8,9] ];

IntArray ia<= mda[0];
i=ia[2];
trace "ia[2]=mda[0][2]="+i;

i=mda[2][1];
trace "mda[2][1]="+i;
```

16.5.1. hash of hashes

To create the infamous "*hash of hashes*", no pointer arrays are required. The HashTable class can store arbitrary datatypes (including pointers) therefore the declaration and handling of multidimensional hash tables is fairly straight forward:

Example:

```
HashTable mht <= #[
                      "ht1"=#["one"=1, "two"=2.2],
                      "ht2"=#["three"=3, "four"=4.4]
                  ];

HashTable ht <= mht["ht2"];

trace mht["ht1"]["two"];
```

Note:

Similar to array initializers, the #[] hash table initializer returns constant objects.
Nevertheless, when an initialization expression is called more than once, the expression lists are evaluated again and the elements of the hash table are updated.

16.6. Pools

A `Pool` is a container for uniform objects which are not accessed sequentially but rather using so called *name-IDs*. The main advantage rests with the fast `qAlloc()` and `qFree()` methods; a linear search for free elements is not required even after many nested alloc/free calls.

Example:

```
Pool p; p.template=String; p.alloc(10);
int nameid; loop(10) {
   nameid=p.qAlloc();
   p[nameid]="test"+rnd(1024);
}
foreach nameid in p trace "p["+nameid+"]="+p[nameid];
```

Example:

```
class MyPoolEntry {
      String name;
}

function main() {
      Pool pool;
      int i;
      int id;
      MyPoolEntry ce;
      String s;

      pool.template=MyPoolEntry;
      pool.alloc(32);
      loop(10)
      {
            id=pool.qAlloc();
            ce<=pool[id];
            ce.name="poolentry_"+id;
      }
      i=0;
      foreach id in pool
      {
            trace("foreach qFree i=="+i);
            if(++i<5)
            {
                  // detach entry, does not affect iterated list
                  pool.qFree(id);
            }
            else
                  id=-1; // end foreach
      }
      foreach id in pool
      {
            ce<=pool[id];
            s<=ce.name;
            trace("foreach entry \""+s+"\"\n");
      }
      pool.free();
}
```

16.7. Trees

A tree is basically a special form of a double-linked list and consists of `TreeNodes` which are linked to their predecessor (*parent*), to nodes on the same hierarchy level ("*left*") resp. to branching off sub-trees ("*right*"). Each TreeNode may furthermore store an object pointer to reference abitrary user data.
An example for a tree structure imported from an `XML` file is given on p.26.

16.8. Stacks

`TKS` does not support recursive function calls since both global and local variables are static, i.e. located at fixed memory addresses. Since recursion makes programming a lot more comfortable in certain cases, e.g. when writing a `HTML` preprocessor, the following *trick* can be used:

By moving all local variables into a user defined class (*stackframe*) which is instanciated every time the function is called, the problem is avoided that a local variable must not be used more than once at at time. The *stackframe* objects are managed by the `Stack` class which provides the necessary `push()` and `pop()` methods.

Example:

```
class FibStackFrame {
    int i;
}

Stack stFib;
stFib.init(FibStackFrame, 8);

function Fib(int _i) {
     int r;
     if(_i==0)
          return 0;
     else
          if(_i==1)
               return 1;
          else
          {
               // ---- get new function stackframe ----
               FibStackFrame st<=stFib.push(); st.i=_i;

               // the Fib calls overwrite function variable _i
               r= Fib(st.i-1) + Fib(st.i-2);

               // restore old function stackframe
               st<=stFib.pop();

               return r;
          }
}

trace("fib(9)="+Fib(9));
```

17. The Just In Time (*JIT*) compiler

The JIT compiler speeds up the execution of frequently called script statement sequences.

When developing (prototypes for) "*realtime*" multimedia applications or applications that need to evaluate larger datasets, the JIT compiler may help to improve the response time of such a system significantly.

The JIT compiler translates statement blocks which have been opened using the compile keyword to an internal bytecode (*p-code*) representation. This *p-code* is optimized by some post processing routines and then finally translated to native CPU code.

By using a *CPUTable* to emit the actual machine code, most of the platform–dependent code is removed from the JIT compiler classes at source level. The *CPUTable* contains the *micro-programs* for all *p-code* instructions. In order to add support for other CPU architectures, this table can be exchanged for a more suitable set of *micro programs*.

The TKS compiler will replace every *p-code* of a JIT program by its respective *micro-program* in order to finally generate native machine code.

Variables, constants and jump/function call addresses in the native code are later replaced by their actual values with the help of some magic (32Bit) place holder symbols that are stored in the *CPUTable*:

"**magic**" **symbol**	**instruction**
0xC0DE1234 :	halt
0xC0DEC0DE :	bra, sfcmp, sicmp, sitestz, sitestzp, siloop
0xC0FFC0FE :	movvc
0xC0FFC4C4 :	pushv, pushivinc, pushinciv, pushivdec, pushdeciv, popv, movv, movvc, inciv,
0xC0FFC4C4	deciv, pushv
0xC0FFC0FF :	pushc, loadc,
0xC0FFC4C5 :	movv
0xC0FFAAA1 :	iasel, iaselbc
0xC0FFABA1 :	iaselbc
0xC0FFAAA2 :	fasel
0xC0FFABA2 :	faselbc
0xC0FFC7C7 :	incliv, decliv, apushl, apushlbc, apushw, apushwbc, apushb, apushbbc, apush4b, apush4bbc, apushnextl, apushnextlbc, apushnextw, apushnextwbc, apushnextb, apushnextbbc, apushnext4b, apushnext4bbc, apushl16, apushl16bc, apushw16, apushw16bc, apushb16, apushb16bc, apush4b16, apush4b16bc, apopl, apoplbc, apopw, apopwbc, apopb, apopbbc, apop4b, apop4bbc, apopnextl, apopnextlbc, apopnextw, apopnextwbc, apopnextb, apopnextbbc, apopnext4b, apopnext4bbc, apopl16, apopl16bc, apopw16, apopw16bc, apopb16, apopb16bc, apop4b16, apop4b16bc, apushl2, apushl2bc, apushl3, apushl3bc, apushl4, apushl4bc, apushnextl2, apushnextl2bc, apushnextl3, apushnextl3bc, apushnextl4, apushnextl4bc, apushl216, apushl216bc, apushl316, apushl316bc, apushl416, apushl416bc, apopl2, apopl2bc, apopl3, apopl3bc, apopl4, apopl4bc, apopnextl2, apopnextl2bc, apopnextl3, apopnextl3bc, apopnextl4, apopnextl4bc, apopl216, apopl216bc, apopl316, apopl316bc, apopl416, apopl416bc
0xC0FFC7C8 :	apushl16, apushl16bc, apushw16, apushw16bc, apushb16, apushb16bc, apush4b16, apush4b16bc, apop16, apopl16bc, apopw16, apopw16bc, apopb16, apopb16bc, apop4b16, apop4b16bc, apushl216, apushl216bc, apushl316, apushl316bc, apushl416, apushl416bc, apopl216, apopl216bc, apopl316, apopl316bc, apopl416, apopl416bc

17.1. Usage

The programmer must tag the statement blocks which are to be compiled to native code by opening them using the keyword `compile`.

Please see p.58 for some hints on usage and limitations.

Example:

```
int flags[8191]; flags.numElements=8191;

compile
{
	int size= 8190;
	int i, prime, k, count, iter;
	trace "Eratosthenes Sieve prime number calculation";
	trace "10 iterations<br>";

	for (iter = 1; iter <= 10; iter++)
	{
		count = 0;

		for (i = 0; i <= size; i++) flags[i] = true;

		for (i = 0; i <= size; i++)
		{
			if (flags[i])
			{
				prime = i + i + 3;
				k = i + prime;
				while (k <= size)
				{
					flags[k] = false;
					k += prime;
				}
				count ++;
			}
		}
	}
	trace count + " primes";
}
```

which compiles to the following bytecode ("sieve.tks" disassembly page):

```
(#0000)00000000: pushc 8190 (0.000000f);    // "sieve.tks" disassembly ("tks -da ...")
(#0001)00000002: popv size;                 // left: memory adr (#dword) right: p-code
(#0002)00000004: pushc 9052904 (0.000000f);
(#0003)00000006: loadc 4417431 (0.000000f);
(#0004)00000008: apicall;
(#0005)00000009: incstp;
(#0006)0000000a: pushc 9053024 (0.000000f);
(#0007)0000000c: loadc 4417431 (0.000000f);
(#0008)0000000e: apicall;
(#0009)0000000f: incstp;
(#0010)00000010: pushc 1 (0.000000f);
(#0011)00000012: popv iter;
(#0012)00000014: pushv iter;
(#0013)00000016: pushc 10 (0.000000f);
(#0014)00000018: sicmpb <=;
(#0015)00000019: sitestz 0079;
(#0016)0000001b: pushc 0 (0.000000f);
(#0017)0000001d: popv count;
(#0018)0000001f: pushc 0 (0.000000f);
(#0019)00000021: popv i;
(#0020)00000023: pushv i;
(#0021)00000025: pushv size;
(#0022)00000027: sicmpb <=;
(#0023)00000028: sitestz 0035;
(#0024)0000002a: pushc 1 (0.000000f);
(#0025)0000002c: pushv i;
(#0026)0000002e: iasel 0;
(#0027)00000030: siapopl;
(#0028)00000031: inciv i;
(#0029)00000033: bra 0023;
(#0030)00000035: pushc 0 (0.000000f);
(#0031)00000037: popv i;
(#0032)00000039: pushv i;
(#0033)0000003b: pushv size;
(#0034)0000003d: sicmpb <=;
(#0035)0000003e: sitestz 0075;
(#0036)00000040: pushv i;
(#0037)00000042: iasel 0;
(#0038)00000044: siapushl;
(#0039)00000045: sitestz 0071;
(#0040)00000047: pushv i;
(#0041)00000049: pushv i;
(#0042)0000004b: siadd;
(#0043)0000004c: pushc 3 (0.000000f);
(#0044)0000004e: siadd;
(#0045)0000004f: popv prime;
(#0046)00000051: pushv i;
(#0047)00000053: pushv prime;
(#0048)00000055: siadd;
(#0049)00000056: popv k;
(#0050)00000058: pushv k;
(#0051)0000005a: pushv size;
(#0052)0000005c: sicmpb <=;
(#0053)0000005d: sitestz 006f;
(#0054)0000005f: pushc 0 (0.000000f);
(#0055)00000061: pushv k;
(#0056)00000063: iasel 0;
(#0057)00000065: siapopl;
(#0058)00000066: pushv k;
(#0059)00000068: pushv prime;
(#0060)0000006a: siadd;
(#0061)0000006b: popv k;
(#0062)0000006d: bra 0058;
(#0063)0000006f: inciv count;
(#0064)00000071: inciv i;
(#0065)00000073: bra 0039;
(#0066)00000075: inciv iter;
(#0067)00000077: bra 0014;
(#0068)00000079: pushc 8991272 (0.000000f);
(#0069)0000007b: loadc 4417431 (0.000000f);
(#0070)0000007d: apicall;
(#0071)0000007e: incstp;
(#0072)0000007f: halt;
```

17.2. Limitations and hints

- the `return` statement may not be used
- not all control structures can be compiled to native code. A *hybrid execution* mixes `C` function calls with native code (`apicall` p-code).
- There following statements/expressions can be compiled to native machine code:
  - *ARGBExpr*
  - *BuiltinFun* (*sin, cos, tan, sqrt, rad, deg, abs, frac, tcint, tcfloat, round, rnd*)
  - *Call* (*C++ method call statement*)
  - *ClassMemberExpr*
  - *ClassMemberOpAssign*
  - *ConstVal*
  - *DoubleArgExpr*
  - *DoWhile*
  - *ECall* (*C++ method call expression*)
  - *ExtFunctionExpr* (*C function call*)
  - *For*
  - *IfElse*
  - *IndexedVarExpr* (*int and float only*)
  - *IndexedVarAssign* (*int and float only*)
  - *LoopStatement*
  - *OpObjectMemberAssign*
  - *OpVarAssign*
  - *PointerAssign*
  - *PostDecVarExpr*
  - *PostDecVarStat*
  - *PostIncVarExpr*
  - *PostIncVarStat*
  - *PreDecVarExpr*
  - *PreDecVarStat*
  - *PreIncVarExpr*
  - *PreIncVarStat*
  - *RGBExpr*
  - *SingleArgExpr*
  - *UnresolvedModuleMemberExpr*
  - *VarAssign*
  - *VarExpr*
  - *While*

- the command line option `--forceint` is used to turn off the `JIT` compiler. Debug tools like e.g. pointer validation are only available when using the interpreted mode.
- The `JIT` compiler can access script-class members but not methods.
- *Array boundary checks* can be turned off by using the command line switch `--noboundarycheck`. This can speed up critical loops by up to 30+%.
- Array variables may not be (re-)allocated within a compiled block (`JIT` program) if they are used in the same block
- The current array pointer is stored in an internal structure each time a `JIT` program is called, so that changes to the array pointers are not recognized until the next program call
- the `JIT` compiler is currently only available for `80x86` compatible `CPUs`
- the `JIT` cpu-table is produced by a `TKS` script that generates `C` header and source files (basically consisting of inline assembler statements).
- The byte/word array instructions are currently unused
- The `-da` command line switch turns on the disassembler

17.3. P-Code instruction set

The following intermediate byte code instruction set is used by the JIT compiler to translate source code to native CPU code. Arguments are usually passed on the hardware stack. Script classes require a static array to keep track of nested class calls. The list may be a little outdated but basically nothing has gravely changed.

```
halt                    terminate the VM.
bra <label>             branch to user defined label
sfcmp <relop> <label>
                        compare float values from stack and branch to label if
                        comparison evaluates to 1
sicmp <relop> <label>
                        compare int values from stack and branch to label
                        if comparison evaluates to 1
sitestz <label>         pop int from stack, compare with 0 and branch to label if 1
sitestzp <label>        compare int value from stack and branch to label
                        if comparison evaluates to 1
bsr <label>             branch to subroutine.
rts                     return from subroutine
siloop <label>          decrement int value from stack and branch to label
                        if value>0
swaps                   swap upper stack values
pushlv <lvar>           push local variable lvar onto stack
pushlivinc <lvar>       push local int variable lvar onto stack and increment it
pushincliv <lvar>       increment and push local int variable lvar onto stack
pushlivdec <lvar>       push local int variable lvar onto stack and decrement it
pushdecliv <lvar>       decrement and push local int variable lvar onto stack
poplv <lvar>            pop local variable lvar from stack
pushv <var>             push global/function variable var onto stack
pushivinc <var>         push global/function int variable var onto stack and
                        increment it
pushinciv <var>         increment and push global/function int variable var onto stack
pushivdec <var>         push global/function int variable var onto stack and
                        decrement it
pushdeciv <var>         decrement and push global/function int variable var onto stack
popv <var>              pop global/function variable var from stack
pushc <const>           push constant value const onto stack
movlv <lvar> <lvar2>    move content of lvar2 to lvar
movlvc <lvar> <const>   move constant value const to lvar
movv <var> <var2>       move content of var2 to var
movvc <var> <const>     move constant value const to var
livandliv <lvar> <lvar2>
                        and local int var lvar with local int var lvar2
livorliv <lvar> <lvar2>
                        or local int var lvar with local int var lvar2
liveorliv <lvar> <lvar2>
                        eor/xor local int var lvar with local int var lvar2
livmodliv <lvar> <lvar2>
                        store lvar mod lvar2 in local int variable lvar
livaddliv <lvar> <lvar2>
                        add local int var lvar2 to local int var lvar
livsubliv <lvar> <lvar2>
                        sub local int var lvar2 from local int var lvar
livmulliv <lvar> <lvar2>
                        mul local int var lvar with local int var lvar2
livdivliv <lvar> <lvar2>
                        div local int var lvar by local int var lvar2
```

lfvaddlfv *<lvar> <lvar2>*
add local float var lvar2 to local float var lvar
lfvsublfv *<lvar> <lvar2>*
sub local float var lvar2 from local float var lvar
lfvmullfv *<lvar> <lvar2>*
mul local float var lvar with local float var lvar2
lfvdivlfv *<lvar> <lvar2>*
div local float var lvar by local float var lvar2
pushlivandliv *<lvar> <lvar2>*
and local int var lvar with local int var lvar2 and push result onto stack
pushlivorliv *<lvar> <lvar2>*
or local int var lvar with local int var lvar2 and push result onto stack
pushliveorliv *<lvar> <lvar2>*
eor/xor local int var lvar with local int var lvar2 and push result onto stack
pushlivmodliv *<lvar> <lvar2>*
push lvar mod lvar2 onto stack
pushlivaddliv *<lvar> <lvar2>*
add local int var lvar2 to local int var lvar and push result onto stack
pushlivsubliv *<lvar> <lvar2>*
sub local int var lvar2 from local int var lvar and push result onto stack
pushlivmulliv *<lvar> <lvar2>*
mul local int var lvar with local int var lvar2 and push result onto stack
pushlivdivliv *<lvar> <lvar2>*
div local int var lvar by local int var lvar2 and push result onto stack
pushlivaslliv *<lvar> <lvar2>*
shift left local int var lvar by local int var lvar2 bits and push result onto stack
pushlivasrliv *<lvar> <lvar2>*
shift right local int var lvar by local int var lvar2 bits and push result onto stack
pushlfvaddlfv *<lvar> <lvar2>*
add local float var lvar2 to local float var lvar and push result onto stack
pushlfvsublfv *<lvar> <lvar2>*
sub local float var lvar2 from local float var lvar and push result onto stack
pushlfvmullfv *<lvar> <lvar2>*
mul local float var lvar with local float var lvar2 and push result onto stack
pushlfvdivlfv *<lvar> <lvar2>*
div local float var lvar by local float var lvar2 and push result onto stack
livandc *<lvar> <const>*
and local int var lvar with constant value const.
livorc *<lvar> <const>*
or local int var lvar with constant value const.
liveorc *<lvar> <const>*
eor/xor local int var lvar with constant value const.
livmodc *<lvar> <const>*
lvar mod const.
livaddc *<lvar> <const>*
add constant value const to local int var lvar.

livsubc <*lvar*> <*const*>
sub constant value const from local int var lvar.
livmulc <*lvar*> <*const*>
mul local int var lvar with constant value const.
livdivc <*lvar*> <*const*>
div local int var lvar by constant value const.
livaslc <*lvar*> <*const*>
shift left local int var lvar by const bits.
livasrc <*lvar*> <*const*>
shift right local int var lvar by const bits.
pushlivandc <*lvar*> <*const*>
and local int var lvar with constant value const and
push result onto stack
pushlivorc <*lvar*> <*const*> *or local int var lvar with constant value* const *and*
push result onto stack
pushliveorc <*lvar*> <*const*>
eor/xor local int var lvar with constant value const and
push result onto stack
pushlivmodc <*lvar*> <*const*>
push lvar mod const onto stack
pushlivaddc <*lvar*> <*const*>
add constant value const to local int var lvar and
push result onto stack
pushlivsubc <*lvar*> <*const*>
sub constant value const from local int var lvar and
push result onto stack
pushlivmulc <*lvar*> <*const*>
mul local int var lvar with constant value const and
push result onto stack
pushlivdivc <*lvar*> <*const*>
div local int var lvar by constant value const and
push result onto stack
pushlivaslc <*lvar*> <*const*>
shift left local int var lvar by const bits and
push result onto stack
pushlivasrc <*lvar*> <*const*>
shift right local int var lvar by const bits and
push result onto stack
lfvaddc <*lvar*> <*const*>
add constant value const to local float var lvar.
lfvsubc <*lvar*> <*const*>
sub constant value const from local float var lvar.
lfvmulc <*lvar*> <*const*>
mul local float var lvar with constant value const.
lfvdivc <*lvar*> <*const*>
div local float var lvar by constant value const.
pushlfvaddc <*lvar*> <*const*>
add constant value const to local float var lvar and
push result onto stack
pushlfvsubc <*lvar*> <*const*>
sub constant value const from local float var lvar and
push result onto stack
pushlfvmulc <*lvar*> <*const*>
mul local float var lvar with constant value const and
push result onto stack
pushlfvdivc <*lvar*> <*const*>
div local float var lvar by constant value const and
push result onto stack
sinot *C-like logical not operator. pops int val from stack and pushes either 0 or 1*
siinv *bitwise not int val from stack*

sineg	*change sign of stack int val*
siquad	*multiply stack int val with itself*
sitestbz	*check if stack int val >0 and set it to either 0 or 1*
sitestbz2	*check if stack+1 int val >0 and set it to either 0 or 1*
sfneg	*change sign of stack float val*
sfquad	*multiply stack float val with itself*
sfpow	*st[0]= st[0] raised to the power of st[1]*
sfsin	*calc sin of stack float val (rad)*
sfcos	*calc cos of stack float val (rad)*
sftan	*calc tan of stack float val (rad)*
sfatan2	*calc arc tan of st[1]/st[0] (both float)*
sfsqrt	*calc square root of stack float val*
sirnd	*push new random int val onto stack*
siabs	*calc absolute value of stack int val*
sfabs	*calc absolute value of stack float val*
sffrac	*calc remainder of stack float val*
sfround	*round stack float val*
siand	*bitwise and stack int values*
sior	*bitwise* or stack int values
sieor	*bitwise eor/xor stack int values*
simod	*sp[0]=sp[1]%sp[0]*
siadd	*add stack int values*
sisub	*sp[0]=sp[1]-sp[0]*
simul	*multiplicate stack int values*
sidiv	*sp[0]=sp[1]/sp[0]. division by zero is not caught*
siasl	*sp[0]=sp[1]<<sp[0]*
siasr	*sp[0]=sp[1]>>sp[0]*
sicmpb *<relop>*	*compare stack int values and store result (1,0)*
sfadd	*add stack float values*
sfsub	*sp[0]=sp[1]-sp[0]*
sfmul	*multiplicate stack float values*
sfdiv	*sp[0]=sp[1]/sp[0]*
sfcmpb *<relop>*	*compare stack float values and store result (1,0)*
stcif	*typecast stack int val to float*
stcif2	*typecast stack+1 int val to float*
stcfi	*typecast stack float val to int*
stcfi2	*typecast stack+1 float val to int*
iasel *<var>*	*load int array variable var into array base register*
fasel *<var>*	*load float array variable var into array base register*
apushl *<lvar>*	*push dword from array at offset stored in lvar onto stack*
siapushl	*pop offset and push dword from array onto stack*
apushl2 *<lvar>*	*push 2 dwords from array at offset stored in lvar onto stack*
apushl3 *<lvar>*	*push 3 dwords from array at offset stored in lvar onto stack*
apushl4 *<lvar>*	*push 4 dwords from array at offset stored in lvar onto stack*
apushw *<lvar>*	*push word from array at offset stored in lvar onto stack. the word is expanded to dword.*
siapushw	*pop offset, push word from array onto stack. the word is expanded to dword.*
apushb *<lvar>*	*push byte from array at offset stored in lvar onto stack. the byte is expanded to dword.*
siapushb	*pop offset, push byte from array onto stack. the byte is expanded to dword.*
apush4b *<lvar>*	*push 4 bytes from array at offset stored in lvar onto stack. the bytes are expanded to dwords.*
siapush4b	*pop offset, push 4 bytes from array onto stack. the bytes are expanded to dwords.*
apushnextl *<lvar>*	*push dword from array at offset stored in lvar onto stack and increase offset*
apushnextl2 *<lvar>*	*push 2 dwords from array at offset stored in lvar onto stack and increase offset*

apushnextl3 *<lvar>* — *push 3 dwords from array at offset stored in lvar onto stack and increase offset*

apushnextl4 *<lvar>* — *push 4 dwords from array at offset stored in lvar onto stack and increase offset*

apushnextw *<lvar>* — *push word from array at offset stored in lvar onto stack and increase offset. the word is expanded to dword.*

apushnextb *<lvar>* — *push byte from array at offset stored in lvar onto stack and increase offset. the byte is expanded to dword.*

apushnext4b *<lvar>* — *push 4 bytes from array at offset stored in lvar onto stack and increase offset. the bytes are expanded to dwords.*

apushl16 *<lvar_off> <lvar_add>* — *push dword from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off*

apushl216 *<lvar_off> <lvar_add>* — *push 2 dwords from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off*

apushl316 *<lvar_off> <lvar_add>* — *push 3 dwords from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off*

apushl416 *<lvar_off> <lvar_add>* — *push 4 dwords from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off*

apushw16 *<lvar_off> <lvar_add>* — *push word from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off. the word is expanded to dword.*

apushb16 *<lvar_off> <lvar_add>* — *push byte from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off. the byte is expanded to dword.*

apush4b16 *<lvar_off> <lvar_add>* — *push 4 bytes from array at offset lvar_off>>16 onto stack. add lvar_add to lvar_off. the bytes are expanded to dword.*

apopl *<lvar>* — pop dword and store in array at offset lvar

siapopl — *pop offset, pop dword and store in array*

apopl2 *<lvar>* — *store 2 stack dwords in array at offset lvar. the bottom element will be stored at offset+1.*

apopl3 *<lvar>* — *store 3 stack dwords in array at offset lvar. the bottom element will be stored at offset+2.*

apopl4 *<lvar>* — *store 4 stack dwords in array at offset lvar. the bottom element will be stored at offset+3.*

apopw *<lvar>* — *pop dword, shrink to word and store in array at offset lvar.*

siapopw — *pop offset, pop dword, shrink to word and store in array.*

apopb *<lvar>* — *pop dword, shrink to byte and store in array at offset lvar*

siapopb — *pop offset, pop dword, shrink to byte and store in array*

apop4b <lvar> — *pop 4 dwords, shrink to bytes and store in array at offset lvar*

siapop4b — *pop offset, pop 4 dwords, shrink to bytes and store in array*

apopnextl *<lvar>* — *pop dword and store in array at offset lvar. increase offset.*

apopnextl2 *<lvar>* — *pop 2 dwords and store in array at offset lvar. increase offset.*

apopnextl3 *<lvar>* — *pop 3 dwords and store in array at offset lvar. increase offset.*

apopnextl4 *<lvar>* — *pop 4 dwords and store in array at offset lvar. increase offset.*

apopnextw *<lvar>* — *pop dword, shrink to word and store in array at offset lvar. increase offset.*

apopnextb *<lvar>* — *pop dwords, shrink to byte and store in array at offset lvar. increase offset.*

apopnext4b *<lvar>* — *pop 4 dwords, shrink to bytes and store in array at offset lvar. increase offset.*

apopl16 *<lvar_off> <lvar_add>* — *pop dword and store in array at offset lvar_off>>16. add lvar_add to lvar_off*

```
apopl216 <lvar_off> <lvar_add>
                        pop 2 dwords and store in array at offset lvar_off>>16.
                        add lvar_add to lvar_off
apopl316 <lvar_off> <lvar_add>
                        pop 3 dwords and store in array at offset lvar_off>>16.
                        add lvar_add to lvar_off
apopl416 <lvar_off> <lvar_add>
                        pop 4 dwords and store in array at offset lvar_off>>16.
                        add lvar_add to lvar_off
apopw16 <lvar_off> <lvar_add>
                        store stack word in array at offset lvar_off>>16.
                        add lvar_add to lvar_off
apopb16 <lvar_off> <lvar_add>
                        store stack byte in array at offset lvar_off>>16.
                        add lvar_add to lvar_off
apop4b16 <lvar_off> <lvar_add>
                        store stack 4 bytes in array at offset lvar_off>>16.
                        add lvar_add to lvar_off
otswitch <voidlabel> <intlabel> <floatlabel> <objectlabel>
                        pop variable return type (0,1,2,3) and
                        branch to label according to type
sipush[0..3]            push constant int val [0..3] onto stack
sipop                   pop int/float val from stack and discard it.
siswapw                 swap upper and lower word of int stack value
sipackargb              pop 4 (byte) ints from stack and push packed 32bit int
siunpackargb            pop packed 32bit int from stack and push 4 (byte) ints
simk16                  pop float from stack and push 16:16 fixed point int
                        suitable for array access
```

18. The *YAC* C/C++ plugin interface

YAC, short for ***Y**et **A**nother **C**omponent Object Model*, provides an interface to dynamically loaded C, C++ and Assembler components (dynamic link libraries / shared objects).

From a technical point of view, the YAC project is independent from TKS since YAC plugins need to have little knowledge of the script engine interna. Please notice that any other plugin host (e.g. scriptengine or application) which implements the YAC interface could easily use the classes, functions and constants provided by a YAC plugin. Unfortunately, the interoperability across different script engine / host architectures is very difficult with other plugin interfaces since they often require the programmer to write lots of host-dependent code.

18.1. Key features:

- typedefs for scalars (sU8, sU16, sU32, sU64, sUI, sS8, sS16, ...)
- simple datastructures (memptr, Value, Object, String, Buffer, IntArray, FloatArray)
- support for C++ classes and C functions
- parameters are directly passed to called methods (no need to decode script engine stacks)
- parameter signatures are encoded in 4*2 = 8 bits (class methods) resp. 8*2= 16 bits (functions) to be decoded by the C preprocessor, thus no external wrapper generators are required.
- no host link library required (just a couple of header files)
- ► uses the regular C++ runtime library
- C/C++ code can call script functions (e.g. event handlers and the like)
- API class ID enumeration (Object, Integer, Float, String, Class, Event, IntArray, FloatArray, Envelope, StringArray, ClassArray, ObjectArray, PointerArray, HashTable, Pool, Stack, Stream, Buffer, File, ...*<user>*...)
- plugins can access classes and constants provided by other plugins
- array and hashtable interfaces
- serialization interface (to load/store the state of an object (hierarchy)),
- operator interface (arithmetic operations, assignment and initialization)
- I/O stream interface (i.e. plugins can access files opened by the plugin host)
- property interface (e.g. to examine / access user defined classes)

18.2. Basic datastructures:

```
union yac_memptr
      void              *any;
      sChar             *c;
      sS8               *s1;
      sU8               *u1;
      sS16              *s2;
      sU16              *u2;
      sS32              *s4;
      sU32              *u4;
      sF32              *f4;
      sSI               *si;
      sUI               *ui;
      YAC_Object        *o;
      void             **iany;
      sChar            **ic;
      sS8              **is1;
      sU8              **iu1;
      sS16             **is2;
      sU16             **iu2;
      // ....
```

```
class YAC_Value
      union value;
      sU16  deleteme;
      sU16  type; // 0=void, 1=int, 2=float, 3=Object, 4=String
```
Used to pass arguments to user defined functions or return a value to the caller of a C++ method or C function.

```
class YAC_Object
      sUI class_ID;
      static sUI object_counter;
```
Base class for all C++ API classes

```
class YAC_Host
      sU8 cpp_typecast_map[YAC_CPP_MAX_CLASSES][YAC_CPP_MAX_CLASSES]
```
Used for runtime type validation of objects, also provides an interface to register new C++ classes, query/call script functions and instanciate new objects.

```
class YAC_String
      sU32    buflen;
      sU32    key;
      sChar * chars;
      void  * clones; // (StaticList::Node*)
```
Basic wrapper class for char sequences / arrays

```
class YAC_Buffer
      sUI   size;
      sUI   io_offset;
      sU8 * buffer;
```
Generic byte array

```
class YAC_FloatArray
      sBool  own_data;
      sUI    max_elements;
      sUI    num_elements;
      sF32 * elements;
```
Generic wrapper class for float arrays

```
class YAC_IntArray
      sBool  own_data;
      sUI    max_elements;
      sUI    num_elements;
      sSI  * elements;
```
Generic wrapper class for int arrays

```
class YAC_Event
      sU32       time_stamp
      YAC_String data;
```

The HTML formatted source of the yac.h plugin interface header file can be found here:
http://tkscript.de/4s/yac.h.html.

The software development kit can be downloaded here:
http://tkscript.de/files/yac.zip

18.3. The host interface

TKS includes a plugin loader which implements the YAC interface described in this chapter. The loader passes the current instance of the TKScriptEngine class (which is derived from the YAC_Host class) to an extension library when the library is requested by a script :

```
use <library>;
```

The associated <library>.dll resp. <library.so> file is openend in the current working directory, or the directory specified by the TKS_PLUGIN_PATH variable.

When the application host (i.e. the TKscript engine) loads an extension it looks for the *magic* symbols

```
void YAC_Init(YAC_Host *_host)
void YAC_Exit(YAC_Host *_host)
```

which represent the entry and exit points of an extension library.

A YAC_Host object pointer is passed to the YAC_Init() and YAC_Exit() functions of a library.

The extension may use the YAC_Host *_host object to:

- Register new classes (yacRegisterClass())
- Retrieve information about class inheritage (cpp_typecast_map array)
- Allocate and free Objects (yacNew(), yacDelete(), yacNewByID())
- Output debug messages (yacPrint(), printf(), yacGetDebugLevel())
- Send events to the running application (yacSendUserEvent())
- Call script functions (yacFindFunction(), yacEvalFunction())

18.4. Functions

Another *magic* symbol is used to query the list of functions provided by a plugin:

```
const sChar*YAC_GetFunctionStringList(void)
```

Example:

```
const sChar*YAC_GetFunctionStringList(void) {
      return
      "glAlphaFunc(if) "
      "glArrayElement(i) "
      "glBegin(i) "
      "glBindTexture(ii) "
      "MIDINoteToFrequency(f):f "; // excerpt from the TKSDL plugin
}
```

Function names in the "list" returned by `YAC_GetFunctionStringList()` are separated by the whitespace character (`ASCII` 32). A maximum number of 8 arguments may be passed to a function. Objects and Strings are passed by reference.

The type signature of a function is encoded using the characters `i` (`int`), `f` (`float`) and `o` (`Object`). The return type of a function (if any) is separated by the `:` char.

The plugin loader will prepend the string "`APIC_`" to the name of every function before it looks up the symbol in the extension library. Hence, the `C` code for the functions listed in the above example looks like this:

```
YAC_APIC void APIC_glAlphaFunc(sS32 _a, sF32 _b)
      { ::glAlphaFunc(_a, _b); }
YAC_APIC void APIC_glArrayElement(sS32 _a)
      { ::glArrayElement(_a); }
YAC_APIC void APIC_glBegin(sS32 _a)
      { ::glBegin(_a); }
YAC_APIC void APIC_glBindTexture(sS32 _a, sS32 _b)
      { ::glBindTexture(_a, _b); }
YAC_APIC void APIC_MIDINoteToFrequency(sF32 _note, YAC_Value *_r)
      { /* ... */ YAC_RETF(/*...*/); }
```

As you may have noticed, an additional parameter (`YAC_Value *_r`) is passed to the function `APIC_MIDINoteToFrequency()`. This parameter is used to store the return type and value of a function. Unlike methods, functions may currently not have variable return types but this will probably change in future releases.

The following macros are used to initialize return values:

```
#define YAC_RETI(a)   _r->initInt((sSI)(a))
#define YAC_RETF(a)   _r->initFloat((sF32)(a))
#define YAC_RETO(a,b) _r->initObject((YAC_Object*)(a),b)
#define YAC_RETS(a,b) _r->initString((YAC_String*)(a), b)
```

The boolean parameter `b` determines whether the returned object is read-only or has to be deleted by the plugin host.

18.5. Classes

The script engine can be extended by new datatypes in form of C++ classes. These classes may provide new methods, constants and signals that can be used in scripts.

Vice versa, the plugin classes may access the basic functionality (streams/arrays/script functions) provided by the TKS core API. For example, a File object (which implements the yacStream* interface) can be passed to a plugin method so the method can read resp. write data. Another example was using the Texture class provided by the TKSDL plugin in custom plugins which is possible since the Texture class implements the yacArray* interface.

The YAC SDK contains a C preprocessor script (spread across some header files) which generates an inline class Cmd0..Cmd63 for each method that has been selected by using a number of defines/macros:

18.5.1. C++ - Example:

```
class SimpleFile : public YAC_Object {
public:
	YAC_H(SimpleFile);

	void yacOperator(sSI _cmd, YAC_Object *_robj, YAC_Value *_r);
	void getOffset(YAC_Value *_r);
	void setOffset(sSI _off);
	void seek(sSI _offset, sSI _mode);
	void readBuffer(YAC_Object *_buf,
			sSI _off, sSI _num, sSI _resizebuf,
			YAC_Value *_r);
	void close();

	#define YAC_CLASS SimpleFile
	#define YAC_NUM_COMMANDS 6

	#define YAC_CMD_0_NAME "operator"
	#define YAC_CMD_0_RTTI 13
	#define YAC_CMD_0_RET 4
	#define YAC_CMD_0_SYM yacOperator

	#define YAC_CMD_1_NAME "getOffset"
	#define YAC_CMD_1_RTTI 0
	#define YAC_CMD_1_RET 1
	#define YAC_CMD_1_SYM getOffset

	#define YAC_CMD_2_NAME "setOffset"
	#define YAC_CMD_2_RTTI 1
	#define YAC_CMD_2_SYM setOffset

	#define YAC_CMD_3_NAME "seek"
	#define YAC_CMD_3_RTTI 5
	#define YAC_CMD_3_SYM seek

	#define YAC_CMD_4_NAME "readBuffer"
	#define YAC_CMD_4_RTTI 87
	#define YAC_CMD_4_RET 1
	#define YAC_CMD_4_SYM readBuffer

	#define YAC_CMD_5_NAME "close"
	#define YAC_CMD_5_RTTI 0
	#define YAC_CMD_5_SYM close

	#include <yac_createcommands.h>
};

YAC_C(SimpleFile, "SimpleFile");
```

18.5.2. Constants and macros

YAC_CLASS defines the name of the current C++ class.

YAC_NUM_COMMANDS defines the number of methods in a C++ class.

and for each *n* in [0 <= *n* < 64] :

YAC_CMD_*n*_NAME defines the script name of a method.

YAC_CMD_*n*_RTTI defines the parameter signature of a method.
The signature is calculated by bit-shifting the arguments (0-4 args allowed):
(_p1 << 0) | (_p2 << 2) | (_p3 << 4) | (_p4 << 6)
Where _p1.._p4 must be set to the following values:
0 = void ; 1 = int ; 2 = float ; 3 = Object*

YAC_CMD_*n*_RET defines the return type of a method.
The special value 4 has to be used if the function's return type is variable or if the function returns new (i.e. read/write) objects.

YAC_CMD_*n*_SYM defines the name of the actual C++ method

The **YAC_C**() and **YAC_H**() macros are used to declare/implement the following methods:

```
YAC_Object  *yacNewObject(void);
const sChar *yacClassName(void);
```

The **#include "createcommands.h"** statement will finally process these macros and defines, create inline classes and add the following method:

```
YAC_CommandList *YAC_CLASS::yacGetCommandList(void);
```

The inline classes are (automatically) derived from the YAC_Command class:

```
void YAC_Command::exec(YAC_Object *_o, memptr _args);
void YAC_Command::exec(YAC_Object *_o,
                       memptr _args,
                       YAC_Value *_ret)
                      ;
```

18.5.3. Object templates

A plugin must provide a template object for each C++ class that is bound to the scriptengine. The template is used by the engine to *clone* new objects and query the list of methods, signals and constants.

The following template classes can be used (but are not required) to register a (Static) class:

```
template <class T> class YAC_Template {public:T *ctemplate; YAC_CommandList
*ccommands; public: YAC_Template (void) { ctemplate=new T();
ccommands=ctemplate->yacGetCommandList(); yac_host->yacRegisterClass(ctemplate,
1, ccommands); }
          ~YAC_Template() { delete ccommands; delete ctemplate; }
};

template <class T> class YAC_STemplate {public:T *ctemplate; YAC_CommandList
*ccommands; public: YAC_STemplate (void) { ctemplate=new T();
commands=ctemplate->yacGetCommandList(); yac_host->yacRegisterClass(ctemplate,
0, ccommands); }
          ~YAC_STemplate()     { delete ccommands; delete ctemplate; }
};
```

(The source code for the template classes in included in the
http://tkscript.de/plugins/tksdl.zip source distribution.)

C++ - Example:

```
#include <yac.h>
#include <yac_host.h>

class SimpleFile { /* ... see previous pages ... */ };

YAC_Host *yac_host;

YAC_Template <SimpleFile> *t_SimpleFile;

sUI class_ID_SimpleFile;


void YAC_Init(YAC_Host *_host) {
     yac_host=_host;

     t_SimpleFile = new YAC_Template <SimpleFile>;

     class_ID_SimpleFile = t_SimpleFile ->ctemplate ->class_ID;
}

void YAC_Exit(YAC_Host *_host) {
     delete t_SimpleFile;

     if(yac_host->yacGetDebugLevel())
          yac_host->yacPrint("example::YAC_Exit() finished.\n");
}
```

18.5.4. The `YAC_Object` class

The **`YAC_Object`** class includes the following interfaces:

```
class YAC_API YAC_Object { // 4 bytes

        public:
                sUI class_ID;         /// ---- set by YAC_Host::yacRegisterClass()
          static sUI object_counter;  /// ---- tracks the total number of objects

        public:
```

- **Core Object interface**
  - Query the class name (`yacClassName()`)
  - New object instancation (`yacNewObject()`)
  - Query the list of methods (`yacGetCommandList()`)
  - Query the list of constants (`yacGetConstantStringList()`)
  - Instanciate an iterator for a container object (`yacGetIterator`())

```
virtual const sChar     *yacClassName                (void);
virtual YAC_Object      *yacNewObject                (void);
virtual sBool            yacCopy                     (YAC_Object *_o);
virtual sBool            yacEquals                   (YAC_Object *_o);
virtual sUI              yacCalcSize                 (void);
virtual void             yacToString                 (YAC_String *s);
virtual sBool            yacScan                     (sSI *_ip);
virtual sBool            yacScan                     (sF32 *_fp)
virtual sBool            yacScan                     (sF64 *_dp);
virtual void             yacOperator                 (sSI _cmd, YAC_Object *_robj, YAC_Value *_ret);
virtual void             yacOperatorInit             (YAC_Object *_robj);
virtual void             yacOperatorAssign           (YAC_Object *_robj);
virtual void             yacOperatorAdd              (YAC_Object *_robj);
virtual void             yacOperatorSub              (YAC_Object *_robj);
virtual void             yacOperatorMul              (YAC_Object *_robj);
virtual void             yacOperatorDiv              (YAC_Object *_robj);
virtual void             yacOperatorClamp            (YAC_Object *_min, YAC_Object *_max);
virtual void             yacOperatorWrap             (YAC_Object *_min, YAC_Object *_max);
virtual YAC_Iterator    *yacGetIterator              (void);
virtual YAC_CommandList *yacGetCommandList           (void);
virtual void             yacRegisterSignal           (sUI _id, void *f);
virtual void             yacGetSignalStringList      (YAC_String *_retstring);
virtual void             yacGetConstantStringList    (YAC_String *_retstring);
```

- **Array interface**
  - Read/write/alloc elements in arrays, query array structure (`yacArray*()`)

```
virtual YAC_Object      *yacArrayNew                 (void);
virtual sUI              yacArrayAlloc               (sUI _sx, sUI _sy, sUI _type, sUI _elementbytesize);
virtual void             yacArrayCopySize            (YAC_Object *_arrayobject);
virtual void             yacArraySet                 (sUI _index, YAC_Value *_value);
virtual void             yacArrayGet                 (sUI _index, YAC_Value *_ret);
virtual sUI              yacArrayGetWidth            (void);
virtual sUI              yacArrayGetHeight           (void);
virtual sUI              yacArrayGetElementType      (void);
virtual sUI              yacArrayGetElementByteSize  (void);
virtual sUI              yacArrayGetStride           (void);
virtual void            *yacArrayGetPointer          (void);
virtual void             yacArraySetWidth            (sUI);
```

- **Hashtable interface**
  - Read/write elements in associative arrays (`yacHash*()`)

```
virtual void             yacHashSet                  (YAC_String*_key, YAC_Value *_value);
virtual void             yacHashGet                  (YAC_String*_key, YAC_Value *_retvalue);
```

- **Property class interface**
  - Query the structure of a property class, read/write properties (`yacProperty*()`)

```
virtual sChar           *yacPropertyClassName        (void);
virtual sUI              yacPropertyGetNum           (void);
virtual sUI              yacPropertyGetAccessKeyByIndex (sUI _idx);
virtual sUI              yacPropertyGetAccessKeyByName  (sChar *_name);
virtual sUI              yacPropertyGetType          (sUI _accesskey);
virtual sChar           *yacPropertyGetName          (sUI _accesskey);
virtual void             yacPropertySet              (sUI _accesskey, YAC_Value *_value);
virtual void             yacPropertyGet              (sUI _accesskey, YAC_Value *_retvalue);
```

- **I/O stream interface**
  - serialization

```
        sBool          yacCanDeserializeClass      (YAC_Object *_stream);
virtual void           yacSerializeClassName       (YAC_Object *_ofs);
virtual void           yacSerialize                (YAC_Object *_ofs, sUI _usetypeinfo);
virtual sUI            yacDeserialize              (YAC_Object *_ifs, sUI _usetypeinfo);
```

  - Open/read/write/seek/close/query I/O streams (`yacStream*()`)

```
virtual sSI            yacStreamGetErrorCode       (void);
virtual void           yacStreamGetErrorStringByCode (sSI _code, YAC_Value *_r);
virtual sUI            yacStreamGetByteOrder       (void);
virtual void           yacStreamSetByteOrder       (sUI);
virtual void           yacStreamSeek               (sSI _off, sUI _mode);
virtual sUI            yacStreamGetOffset          (void);
virtual void           yacStreamSetOffset          (sUI);
virtual sUI            yacStreamGetSize            (void);
virtual sBool          yacStreamEOF                (void);
virtual sSI            yacStreamRead               (sU8 *, sUI _num);
virtual sU8            yacStreamReadI8             (void);
virtual sU16           yacStreamReadI16            (void);
virtual sU32           yacStreamReadI32            (void);
virtual sF32           yacStreamReadF32            (void);
virtual void           yacStreamReadObject         (YAC_Object *_dest);
virtual sSI            yacStreamReadString         (YAC_String *_dest, sUI _maxlen);
virtual sSI            yacStreamReadBuffer         (YAC_Buffer *_dest, sUI _off, sUI _num, sBool _resize);
virtual sSI            yacStreamWrite              (sU8 *, sUI _num);
virtual void           yacStreamWriteI8            (sU8);
virtual void           yacStreamWriteI16           (sU16);
virtual void           yacStreamWriteI32           (sS32);
virtual void           yacStreamWriteF32           (sF32);
virtual void           yacStreamWriteObject        (YAC_Object*);
virtual sSI            yacStreamWriteString        (YAC_String *, sUI _off, sUI _num);
virtual sSI            yacStreamWriteBuffer        (YAC_Buffer *_buf, sUI _off, sUI _num);
virtual sBool          yacStreamOpenLogic          (sChar *_name_in_pakfile);
virtual sBool          yacStreamOpenLocal          (sChar *_local_pathname, sBool _write);
virtual void           yacStreamClose              (void);
```

- **Signal interface**
  - Callback interface to script functions (e.g. mouse/keyboard handlers) (`yacRegisterSignal(), yacGetSignalStringList()`)

```
virtual void           yacRegisterSignal           (sUI _id, void *f);
virtual void           yacGetSignalStringList      (YAC_String *_retstring);
```

- **Operators** (`yacOperator*()`) (`YAC_OP_...`)

  `ASSIGN=0,ADD,SUB,MUL,DIV,MOD,SHL,SHR,CEQ,CNE,CLE,CLT,CGE,CGT,AND,OR,EOR,NOT,BITNOT,LAND,LOR,LEOR,NEG,INIT.`

```
// call operator (see YAC_OP_)
virtual void           yacOperator                 (sSI _cmd, YAC_Object *_robj, YAC_Value *_ret);
virtual void           yacOperatorInit             (YAC_Object *_robj);
virtual void           yacOperatorAssign           (YAC_Object *_robj);
virtual void           yacOperatorAdd              (YAC_Object *_robj);
virtual void           yacOperatorSub              (YAC_Object *_robj);
virtual void           yacOperatorMul              (YAC_Object *_robj);
virtual void           yacOperatorDiv              (YAC_Object *_robj);
virtual void           yacOperatorClamp            (YAC_Object *_min, YAC_Object *_max);
virtual void           yacOperatorWrap             (YAC_Object *_min, YAC_Object *_max);
```

18.6. Constants

The `yacGetConstantStringList()` method defined by the `YAC_Object` interface is used to add new constants to the script engine. These constants will be visible in all script modules.

C++ - Example:

```
void TKSEnvelope::yacGetConstantStringList(YAC_String *_c) {
    _c->append(
     "ENV_SH:$0 "
     "ENV_LINEAR:$1 "
     "ENV_COSINE:$2 "
     "ENV_QUADRATIC:$3 "
     "ENV_CUBIC:$4 "
     "ENV_QUINTIC:$5 "
     "ENV_SHRESET:$6 "
     "NUM_ENVELOPE_TYPES:$7 "
     );
}
```

The constants are separated by the whitespace character (`ASCII` 32).

18.7. Properties via `set/get`

Members of `C++` classes are not addressed directly but rather using dedicated `set` and `get` methods. This allows for virtual properties which are calculated on request. Most important, the value ranges of members can be validated when properties are set.

The (script-) name of the set method is based on the prefix "`set`" followed by the name of the property with the first letter capitalized, i.e. the set method for "`numElements`" is "`setNumElements`".

C++ - Example:

```
void YAC_FloatArray::tksinter_getNumElements(YAC_Value *_r) {
     YAC_RETI(num_elements);
}

void YAC_FloatArray::tksinter_setNumElements(sSI _ne) {
     if( (((sUI)_ne)<=max_elements) )
          num_elements=_ne;
}
```

Note:

The `tksinter_` prefix in the method names is not required yet recommended.

18.8. Property classes

Property classes represent a kind of meta class mechanism. An example for a property class is a user defined class (meta class type *MyClass*, object class type `Class`).

An object that is derived from the `YAC_Object` class may implement the following methods so it can be queried by script and/or native code :

- `yacPropertyClassName()` - *Query the property class name, e.g. "MyClass"*
- `yacPropertyGetNum()` - *Query the number of properties (members)*
- `yacPropertyGetAccessKeyByIndex()` - *Return the access key for property `#index`*
- `yacPropertyGetAccessKeyByName()` - *Resolve name to access key*
- `yacPropertyGetType()` - *Query the type (1=int, 2=float, 3=Object) of a property*
- `yacPropertyGetName()` - *Query the name of the property assoc. with an access key*
- `yacPropertySet()` - *Change the value of a property*
- `yacPropertyGet()` - *Query the value of a property*

A property is read resp. written by first querying its access key which is represented by an `unsigned integer`. The structure of an access key depends on the respective object. It may be a simple linear offset, a memory address or a hash key.

The access key can be obtained either by name or by the property index in the range of `0..index..yacPropertyGetNum()`.

This mechanism is also accessible through the `TKS` core API object.

Example:

```
class CClass2 {
    float x,y,z,w;
}

class CClass {
    int i;
    float f;
    String s;
    IntArray ia;
    FloatArray fa;
    StringArray sa;
    HashTable ht;
    Pool p;
    Stack st;
    CClass2 c;
}

CClass cobj;

cobj.i=42;
cobj.f=PI;
cobj.s="hello, world.";
cobj.ia=[1,2,3,4,5,6,7,8];
cobj.fa=[0.1, 0.2, 0.3, 0.4];
cobj.sa=["abc","def","ghi","jkl", "mno", "pqr", "stu", "vwxyz"];
cobj.ht=#["hti"=64, "htf"=2PI, "hts"="*foo*bar*foo*bar*foo*bar*"];
Pool p<=cobj.p; p.template=String;
                p.alloc(10);
                String s<=p[p.qAlloc()];
                s="a pool entry..";
```

```
int prop_num=TKS.getNumProperties(cobj);
trace "class "+TKS.getPropertyClassName(cobj)+" has "+
      prop_num+"  properties.";

int i=0;
loop(prop_num) {
    int ak=TKS.getPropertyAccessKeyByIndex(cobj, i++);
    String prop_val;
    String prop_name=TKS.getPropertyNameByAccessKey(cobj, ak);
    Pointer prop_valp<=null;
    prop_valp<=TKS.getPropertyByAccessKey(cobj, ak);
    if(!prop_valp)
        // int / float property type
        prop_val=TKS.getPropertyByAccessKey(cobj, ak);

    else {
        // Object property type
        String prop_cl=TKS.getClassName(prop_valp);
        switch(prop_cl)
            {
            case "String":
                prop_val="String: "+prop_valp;
                break;
            case "Pool":
                p<=prop_valp;
                prop_val="Pool: numElements="+p.numElements;
                break;
            case "Class":
                Class c;
                c<=prop_valp;
                prop_val="Class: name="+TKS.getClassName(c)+
                                "numProperties="+TKS.getNumProperties(c);
                break;
            case "IntArray":
                IntArray ia<=prop_valp;
                prop_val="IntArray: numElements="+ia.numElements+
                                 " maxElements="+ia.maxElements;
                break;
            case "FloatArray":
                FloatArray fa<=prop_valp;
                prop_val="FloatArray: numElements="+fa.numElements+
                                  " maxElements="+fa.maxElements;
                break;
            case "StringArray":
                StringArray sa<=prop_valp;
                prop_val="StringArray: numElements="+sa.numElements+
                                  " maxElements="+sa.maxElements;
                break;
            case "HashTable":
                HashTable ht<=prop_valp;
                prop_val="HashTable: numElements="+ht.numElements+
                                 " maxElements="+ht.maxElements;
                break;
            case "Stack":
                Stack st<=prop_valp;
                prop_val="Stack: index="+st.index+" size="+st.size;
                break;
            }
    }
    trace "  property #"+i+": name=\""+prop_name+
          "\" value=\""+prop_val+"\".";
}
```

18.7. Signals

The signal interface is used to bind a script callback function to an event of a C++ class. The script function is called by the plugin when an event has been received (e.g. a mouse click or key event).

The C++ class must return a list of strings separated by the whitespace (ASCII 32) character. Each signal entry is composed of the signal name and the parameter RTTI of the associated callback function (as described on p.70).

```
void MyClass::yacGetSignalStringList (YAC_String *_ret)
{
      _ret->visit("sig1:7 sig2:1 sig3:2 sig4:85 sig5:0");
};
```

The above example declares the following signals:

name	***_index***	***parameters***
sig1	0	(Object, int)
sig2	1	(int)
sig3	2	(float)
sig4	3	(int, int, int, int)
sig5	4	(void)

The use statement binds a script function to a named signal once the signals have been added to the script engine and calls the following method to register the script function:

```
void yacRegisterSignal (sUI _index, void *f);
```

The _index parameter contains the enumeration index of the signal in the string "list" returned by yacGetSignalStringList(). The argument f provides a pointer to the script function. The pointer should be stored in an array at an enumerated index so it can later be used like this:

```
void MyClass::raiseSig1(void) {
      void *f=signals[MYCLASS_SIG5];
      if(f) yac_host->yacEvalFunction(f, 0, 0);
}
```

Example:

```
use tksdl; // load TKSDL extension library
function onKeyboard(Key _k) { trace _k.name; }
Viewport.openWindow(640, 480); // open a desktop window
use onKeyboard for SDL.onKeyboard; // bind a signal to a script
SDL.eventLoop(); // wait for events
```

The "use callbacks;" statement scans all registered signals and auto-binds any script function which has the same name as a signal.

Example:

```
use tksdl; // load TKSDL extension library
function onKeyboard(Key _k) { trace _k.name; }
function onMouse(int _x, int _y, int _cstate, int _nstate) {
      trace "mouse moved, x="+_x+" y="+_y+" buttons="+_cstate;
}
Viewport.openWindow(640, 480); // open a desktop window
use callbacks; // bind signals to available callbacks
SDL.eventLoop(); // wait for events
```

18.8. Streams

This interface is used to implement new I/O classes (e.g. a zipfile reader extension) or access local files from plugins (see example below).

`TKS` stream classes (e.g. the `File` and `Buffer` classes) support transparent byte order conversion of multi-byte values (16bit *words*, 32bit *double words*).

The most important methods of the `YAC_Object` stream interface are:

- `yacStreamOpenLogic()` *- Open a file mapped in the project file (read-only)*
- `yacStreamOpenLocal()` *- Open a local file (read/write)*
- `yacStreamSetByteOrder()` *- Set the byte order of the file (little / big endian)*
- `yacStreamRead*()` *- Read bytes/words/ints/floats/strings/buffers*
- `yacStreamWrite*()` *- Write bytes/words/ints/floats/strings/buffers*
- `yacStreamSeek()` *- Change the current file offset*
- `yacStreamGetErrorCode()` *- Query the current I/O status*
- `yacStreamClose()` *- Close the stream*

C++ Example*:*

```
YAC_Object *sio=yac_host->yacNewByID(YAC_CLID_FILE);
sBool isopen;
if(_local)
     isopen=sio->yacStreamOpenLocal("infile", 0);
else
     isopen=sio->yacStreamOpenLogic("infile");

if(isopen)
{
     sio->yacStreamSetByteOrder(YAC_BIGENDIAN);
     sU32 i=sio->yacStreamReadI32();
     sio->yacStreamClose();
}
yac_host->yacDelete(sio);
```

18.9. Serialisierung

Dieser Mechanismus dient dazu (komplexe) Datenstrukturen in einer binären Datei (`Stream`) abzulegen um diese z.B. beim nächsten Programmstart auf einfache Art und Weise wieder herstellen zu können.

Die meisten `TKS` API-Klassen unterstützen die (De-)Serialisierung; eine Besonderheit aber bieten benutzerdefinierte Skriptklassen da diese, nachdem die zu (de-)serialisierenden Elemente durch das `tag` Schlüsselwort markiert worden sind, durch einfachen Aufruf der Streaming Operatoren `<<` und `>>` ebenfalls serialisiert werden können, ohne dass dafür, wie aus anderen Sprachen bekannt, spezielle `serialize()` und `deserialize()` Methoden geschrieben werden müssen.
Es werden jedoch immer nur *schreibbare* Objektreferenzen serialisiert um Endlosschleifen zu vermeiden.

Example:

```
class CTest { tag String s,t,v; tag int i,j,k; }
CTest c;
File f;
f.openLocal("out.txt", IOS_OUT);
f<<c; // binary serialization of a user defined class instance
f.close();
```

Beispiel:

```
class MyClass {
     int i;
     tag HashTable ht;
     tag float f;
}
MyClass c; c.ht=#["i"=42, "f"=PI, "s"="hello, world."]; c.f=1.23;
File f; f.openLocal("serial.txt", IOS_OUT);
f << c; // markierte Elemente serialisieren
f.close();
// nun die serialisierte Struktur wieder deserialisieren
f.openLocal("serial.txt", IOS_IN);
Buffer b; f.readBuffer(b, 0, f.size, true); // File in Buffer laden
MyClass d; d<<b;
trace "d.f="+d.f + " d.ht[i]="+d.ht["i"] +
     " d.ht[f]="+d.ht["f"] + " d.ht[s]="+d.ht["s"];
```

18.10. Arrays

Datenstrukturen:

Die Kern-API Klassen `IntArray` und `FloatArray` ebenso wie z.B. die `Texture` Klasse der `TKSDL` Erweiterung implementieren die generischen `YAC_IntArray` bzw. `YAC_FloatArray` Schnittstellen, welche somit auch von anderen Erweiterungen verwendet werden können ohne dass diese weitere Implementierungsdetails der davon abgeleiteten Klassen kennen.

Die wichtigsten Methoden der `YAC_Object` Schnittstelle bzgl. Arrays sind weiterhin:

- `yacArrayNew()` *- Array Objekt für Datentyp erzeugen*
- `yacArrayAlloc()` *- Arrayelemente allokieren*
- `yacArraySet()` *- ein Arrayelement setzen*
- `yacArrayGet()` *- ein Arrayelement auslesen*
- `yacArrayGetElementType()` *- den Typ (1=int, 2=float, 3=Object) der Arrayelemente erfragen*
- `yacArrayGetElementByteSize()` *- Anz. Bytes/Element erfragen (1=byte, 2=word, 4=dword)*
- `yacArrayGetPointer()` *- Zeiger auf erstes Element eines Arrays erfragen*

18.11. Iteratoren

Iteratoren werden von der `foreach` Anweisung verwendet um die Elemente einer Container-Klasse zu durchlaufen.

Beispiel:

```
YAC_Iterator *TKSHashTable::yacGetIterator(void) {
    return new TKSHashTableIterator(this);
}
```

18.12. Aufruf von Skriptfunktionen

Folgende Methoden der `YAC_Host` Schnittstelle dienen dazu aus Erweiterungen heraus auf Skriptfunktionen zugreifen zu können:

- `yacFindFunction()` *- einen untypisierten Zeiger auf eine Skriptfunktion erfragen*
- `yacEvalFunction()` *- eine mittels yacFindFunction() erfragte Skriptfunktion aufrufen. Die Parameterübergabe erfolgt als Feld von YAC_Value Objekten.*

18.13. Anlegen und Löschen von Objekten

Es ist aus technischen Gründen nicht möglich die Speicherverwaltung von Skript-Objekten zwischen Applikations-Host (Scriptengine) und Erweiterungen zu mischen; aus diesem Grunde müssen folgende Methoden verwendet werden:

- `yacNewByID()` *- ein neues Objekt einer Klasse erzeugen, siehe auch YAC_Host::class_ID*
- `yacNew()` *- ein neues Objekt einer Klasse erzeugen, vorher Namen nach ID auflösen*
- `yacDelete()` *- ein zuvor mittels yacNew*() allokiertes Objekt löschen*

19. API reference

Since the printable documentations of the Application Programmer Interfaces (APIs) has not been finished yet, references to the online docs are given here.

However, the core API documentation is part of this documentation and can be found on the following pages.

Core API documentation:
http://tkscript.de/api/tks/index.html

OpenGL / SDL API documentation:
http://tkscript.de/api/tksdl/index.html

Other 3rd party APIs:
http://tkscript.de/plugins.html

19.1. Buffer

- a simple and general purpose binary buffer that can e.g. be used to read files into.
- this class supports the << serialization operator

Inheritance:

```
Object -> Stream -> Buffer
```

Properties:

`size`	- (read-write), stores the current buffer size in bytes
`byteOrder`	- `BIG_ENDIAN` or `LITTLE_ENDIAN`
`offset`	- current byte offset (stream interface)

Methods:

```
            fillZero()
            free()
int         getByteOrder()
int         getOffset()
int         getSize()
String      getString(int _off, int _len)
int         gunzip(Buffer _src, int _off, int _clen, int _ulen)
int         gzip(Buffer _src, int _off, int _ulen, int _level)
int         peekI8()
int         peekI16()
int         peekI32()
int         peekF32()
            pokeI8(int _val)
            pokeI16(int _val)
            pokeI32(int _val)
            pokeF32(int _val)
            resize(int _size)
            setByteOrder(int _type)
            setOffset(int _off)
            setSize(int _size)
int         setString(int _off, String _s)
```

19.2. ClassArray

Inheritance:

```
Object -> ClassArray
```

Properties:

`numElements`	– return/set number of used elements
`maxElements`	– return number of available elements
`nextFree`	– return pointer to next free element

Methods:

```
alloc(int _num)
empty()
free()
getNextFree()
getNumElements()
getMaxElements()
print()
realloc(int _size)
reverse()
setNumElements(int _num)
```

19.3. Configuration

Inheritance:

```
Object -> Configuration
```

Properties:

```
debugLevel      - return/set current debug level (0..99)
forceInt        - enable/disable the JIT compiler
```

Methods:

```
                setDebugLevel(int _level)
int             getDebugLevel()
int             setForceInt(int _bool)
```

19.4. Envelope

Inheritance:

```
Object -> FloatArray -> Envelope
```

Properties:

```
deltaTime       -
interpolation   -
speed           -
time            -
```

Constants:

ENV_SH, ENV_LINEAR, ENV_COSINE, ENV_QUADRATIC, ENV_CUBIC, ENV_QUINTIC, ENV_SHRESET, NUM_ENVELOPE_TYPES.

Methods:

```
float      get()
float      getDeltaTime()
           getInterpolation()
float      getSpeed()
float      getTime()
           reset()
           setInterpolation(int _type)
           setSpeed(float _speed)
           setTime(float _t)
           tickPrecise(float _prec)
```

19.5. Event

Inheritance:

```
Object -> Event
```

Properties:

```
timestamp       -
string          -
```

Methods:

```
int         getTimestamp()
            setTimestamp(int)
String      getString()
            setString(String _s)
```

19.6. File

Inheritance:

```
Object -> Stream -> File
```

Properties:

```
byteOrder       - BIG_ENDIAN or LITTLE_ENDIAN
object          -
offset          -
size            -
i8              -
i16             -
i32             -
f32             -
```

Constants:

SEEK_BEG, SEEK_CUR, SEEK_END,
ERRINVALIDSEEK, ERRIO, ERRCREATEFILE, ERROPENFILE,
IOS_OUT, IOS_IN, IOS_INOUT.

Methods:

```
                close()
                deserialize(Object _o, int _rtti)
int             eof()
                flush()
int             getByteOrder()
int             getErrorCode()
String          getErrorStringByCode(int _code)
int             getI8()
int             getI16()
int             getI32()
float           getF32()
                getObject(Object _o)
int             getOffset()
int             getSize()
int             isOpen()
                setI8(int _i)
                setI16(int _i)
                setI32(int _i)
                setF32(float _f)
int             open(String _name)
int             openLocal(String _name, int _iosmode)
int             readBuffer(Buffer _b, int _off, int _num,
                           int _resize)
int             readString(String _s, int _num)
                seek(int _mode, int _off)
                serialize(Object _o, int _rtti)
                setByteOrder(int _order)
                setObject(Object _o)
                setOffset(int _off)
int             writeBuffer(Buffer _b, int _off, int _num)
int             writeString(String _s, int _off, int _num)
```

19.7. Float

Inheritance:

```
Object -> Float
```

Properties:

```
value           -
```

Methods:

```
float      getValue()
           setValue(float _f)
String     printf(String _fmt)
```

19.8. FloatArray

Inheritance:

```
Object -> FloatArray
```

Properties:

```
numElements     -
maxElements     -
```

Methods:

```
                add(float _f)
                addEmpty(int _num)
int             alloc(int _num)
                blend(FloatArray _o, float _f)
                copyFrom(FloatArray _src, int _off, int _len,
                         int _doff)
                delete(int _idx)
                empty()
                fill(float _f)
                free()
int             getNumElements()
int             getMaxElements()
                insert(int _idx, float _f)
                print(int _channel)
                realloc(int _num)
                reverse()
                scale(FloatArray _src, float _s)
                setNumElements(int _num)
int             visit(FloatArray _o, int _off, int _len)
```

19.9. HashTable

Inheritance:

```
Object -> HashTable
```

Properties:

```
numElements     -
maxElements     -
```

Methods:

```
int         addInt(String _name, int _i)
int         addFloat(String _name, float _f)
int         addObject(String _name, Object _o)
int         addObjectRef(String _name, Object _o)
int         addString(String _name, String _s)
int         alloc(int _num)
            copyFrom(IntArray _src, int _off, int _len,
                     int _doff)
            delete(String _name)
int         exists(String _name)
            free()
<var>       get(String _name)
int         getNumElements()
int         getMaxElements()
            print(int _channel)
```

19.10. IntArray

Inheritance:

```
Object -> IntArray
```

Properties:

```
numElements     -
maxElements     -
```

Methods:

```
                add(int _i)
                addEmpty(int _num)
int             alloc(int _num)
                delete(int _idx)
                empty()
                fill(int _i)
                free()
int             getNumElements()
int             getMaxElements()
int             insert(int _idx, int _i)
                print(int _channel)
int             realloc(int _num)
                reverse()
                setNumElements(int _num)
                swapByteOrder()
int             visit(IntArray _src, int _off, int _num)
```

19.11. Integer

Inheritance:

```
Object -> Integer
```

Properties:

```
value           -
```

Methods:

```
int         getValue()
            setValue(int _i)
String      printf(String _fmt)
```

19.12. ObjectArray

Inheritance:

```
Object -> ObjectArray
```

Properties:

```
nextFree        -
numElements     -
maxElements     -
template        -
```

Methods:

```
int         alloc(int _num)
            empty()
            free()
Object      getNextFree()
int         getNumElements()
int         getMaxElements()
Object      getTemplate()
            print(int _channel)
int         realloc(int _num)
            reverse()
            setNumElements(int _num)
            setTemplate(Object _t)
```

19.13. PointerArray

Inheritance:

```
Object -> PointerArray
```

Properties:

```
numElements     -
maxElements     -
```

Methods:

```
int         alloc(int _num)
            empty()
int         findPointer(Object _o)
            free()
int         getNumElements()
int         getMaxElements()
            print(int _channel)
int         realloc(int _num)
            reverse()
            setNumElements(int _i)
```

19.14. Pool

Inheritance:

```
Object -> Pool
```

Properties:

```
numElements     -
maxElements     -
template        -
```

Methods:

```
int         alloc(int _num)
            empty()
            free()
int         getIDByObject(Object _o)
int         getNumElements()
int         getMaxElements()
Object      getTemplate()
            setTemplate(Object _t)
int         qAlloc()
            qFree(int _id)
            qFreeByObject(Object _o)
```

19.15. Stack

Inheritance:

```
Object -> Stack
```

Properties:

```
index           -
size            -
template        -
```

Methods:

```
int         getSize()
int         getIndex()
Object      getTemplate()
            init(Object _t, int _size)
Object      pop()
Object      push()
            setIndex(int _idx)
int         setSize(int _size)
            setTemplate(Object _t)
```

19.16. Stream

Inheritance:

```
Object -> Stream
```

Properties:

byteOrder	– BIG_ENDIAN or LITTLE_ENDIAN
errorCode	–
i8	–
i16	–
i32	–
f32	–
offset	–
size	–

Methods:

```
int        deserialize(Object _o, int _rtti)
int        eof()
int        getErrorCode()
int        getI8()
int        getI16()
int        getI32()
float      getF32()
int        getByteOrder()
String     getErrorStringByCode(int _code)
           getObject(Object _o)
int        getOffset()
int        getSize()
int        readBuffer(Buffer _b, int _off, int _num,
                      int _resize)
int        readString(String _s, int _num)
int        serialize(Object _o, int _rtti)
           setByteOrder(int _order)
           setI8(int _i)
           setI16(int _i)
           setI32(int _i)
           setF32(float _f)
           setObject(Object _o)
           setOffset(int _off)
int        writeBuffer(Buffer _b, int _off, int _num)
int        writeString(String _s, int _off, int _num)
```

19.17. String

Inheritance:

```
Object -> String
```

Properties:

`length`	– number of chars (including ASCIIZ)

Methods:

```
int         alloc(int _num)
            append(String _s)
            copy(String _src)
            empty()
int         endsWith(String _s)
            fixLength()
            free()
            freeStack()
int         getc(int _off)
int         getLength()
String      getWord(int _nr)
int         indexOf(String _s, int _startoff)
            insert(int _off, String _s)
int         isBlank()
int         lastIndexOf(String _s)
int         load(String _name, int _b_ascii)
int         loadLocal(String _name, int _b_ascii)
int         numIndicesOf(String _s)
TreeNode    parseXML()
int         patternMatch(String _p)
            print(int _channel)
            putc(int _off, int _c)
int         replace(String _s, String _t)
int         saveLocal(String _name)
int         split(int _char)
int         startsWith(String _s)
            substring(String _d, int _off, int _len)
            toLower()
            toUpper()
            trim()
int         words(int _includeembeddedstrings)
```

19.18. StringArray

Inheritance:

```
Object -> StringArray
```

Properties:

```
numElements     -
maxElements     -
nextFree        -
```

Methods:

```
                add(String _s)
                addEmpty(int _num)
int             alloc(int _num)
int             delete(int _idx)
                empty()
                free()
String          getNextFree()
int             getNumElements()
int             getMaxElements()
int             insert(int _idx, String _s)
                print(int _channel)
int             realloc(int _num)
                reverse()
                setNumElements(int _num)
                sortByLength()
                sortByValue(IntArray _ia, int _casesensitive)
                unset()
```

19.19. StringIterator

Inheritance:

```
Object -> StringIterator
```

Properties:

```
current         -
```

Methods:

```
                begin(String _s)
                end()
String          getCurrent()
                head()
int             next()
int             previous()
int             tail()
```

19.20. Time

Inheritance:

```
Object -> Time
```

Properties:

```
hour            -
min             -
month           -
sec             -
year            -
monthday        -
weekday         -
yearday         -
```

Methods:

```
                calc()
int             getHour()
int             getMin()
int             getMonth()
int             getSec()
int             getYear()
int             getMonthday()
int             getWeekday()
int             getYearday()
                now()
                setHour(int _hour)
                setMin(int _min)
                setMonth(int _month)
                setSec(int _sec)
                setYear(int _year)
```

19.21. TKS

Inheritance:

```
Object -> TKS
```

Properties:

Methods:

```
String     constantToString(int _val, String _prefix)
int        getClassID(Object _o)
String     getClassName(Object _o)
int        getVersion()
String     getVersionString()
           setIntPropertyByName(Object _o,
                                String _property,
                                int _val)
           setFloatPropertyByName(Object _o,
                                  String _property,
                                  float _val)
           setObjectPropertyByName(Object _o,
                                   String _property,
                                   Object _v)
int        getNumProperties(Object _o)
int        getPropertyAccessKeyByName(Object _o, String _name)
int        getPropertyAccessKeyByIndex(Object _o, int _index)
<var>      getPropertyByName(Object _o, String _property)
<var>      getPropertyByAccessKey(Object _o, int _ak)
String     getPropertyClassName(Object _o)
String     getPropertyNameByAccessKey(Object _o, int _ak)
<var>      newObjectByName(String _name)
<var>      newObjectByID(int class_ID)
<var>      stringToConstant(String _const)
```

19.22. TreeNode

Inheritance:

```
Object -> TreeNode
```

Properties:

```
left            -
right           -
parent          -
root            -
name            -
id              -
```

Methods:

```
                deleteObject()
                findByName(String _s, int _depth) (NOP)
                free()
String          getId()
TreeNode        getLeft()
String          getName()
int             getNumNodes()
TreeNode        getParent()
Object          getObject()
TreeNode        getRight()
TreeNode        getRoot()
TreeNode        insertLeft(Object _t)
TreeNode        insertRight(Object _t)
TreeNode        seekByPathName(String _s)
                setId(String _s)
                setLeft(TreeNode _n)
                setName(String _name)
                setNewObject(Object _t)
                setObject(Object _o)
                setRight(TreeNode _n)
                writeToHashTable(HashTable _d)
```

20. Verweise

http://www.tcl.tk/doc/scripting.html
“Scripting: Higher Level Programming for the 21st Century”
- a very sophisticated essay by John K. Ousterhout about "scripting

http://portal.acm.org/citation.cfm?id=857076.857077&dl=GUIDE&dl=ACM
- Literaturverweise zum Thema Just-In-Time Compiler

http://java.sun.com
“The source for Java Technology”

http://www.microsoft.com
- C# und die .NET Umgebung

http://www.gnu.org
“the GNU Project and the Free Software Foundation”

http://www.perl.com
“Practical Extraction and Report Language”

http://www.php.net/
“PHP is an HTML-embedded scripting language“

http://www.tkscript.de
“a portable glue script language for C and C++ frameworks”

http://www.opengl.org
“High Performance 2D/3D Graphics”

http://www.khronos.org/opengles/spec.html
“OpenGL ES specification”

http://www.libsdl.org
“Simple DirectMedia Layer”

http://www.fox-toolkit.org/
“a C++ based Toolkit for developing Graphical User Interfaces easily and effectively“

http://www.mysql.com/
“The world's most popular Open Source Database”

http://www.libpng.org/pub/png/
“A Turbo-Studly Image Format with Lossless Compression”

http://www.gzip.org/zlib/
“A Massively Spiffy Yet Delicately Unobtrusive Compression Library”

http://www.scene.org
”a non-profit organization aimed at providing the 'electronic art scene' with a forum for communication”

Appendix A (*ASCII table)*

source: http://www.bbsinc.com/iso8859.html

control characters:

description	***abbrev***	***dec***	***hex***	***description***	***abbrev***	***dec***	***hex***
null	NUL	0	0	start of heading	SOH	1	1
start of text	STX	2	2	end of text	ETX	3	3
end of transmission	EOT	4	4	enquiry	ENQ	5	5
acknowledge	ACK	6	6	bell	BEL	7	7
backspace	BS	8	8	horizontal tab	HT	9	9
line feed, new line	LF, NL	10	A	vertical tab	VT	11	B
form feed, new page	FF, NP	12	C	carriage return	CR	13	D
shift out	SO	14	E	shift in	SI	15	F
data link escape	DLE	16	10	device control 1	DC1	17	11
device control 2	DC2	18	12	device control 3	DC3	19	13
device control 4	DC4	20	14	negative acknowledge	NAK	21	15
synchronous idle	SYN	22	16	end of trans. block	ETB	23	17
cancel	CAN	24	18	end of medium	EM	25	19
substitute	SUB	26	1A	escape	ESC	27	1B
file separator	FS	28	1C	group separator	GS	29	1D
record separator	RS	30	1E	unit separator	US	31	1F

Special characters and numbers:

32, $20	!33, $21	"34, $22	#35, $23	$36, $24	%37, $25	&38, $26	'39, $27
(40, $28	)41, $29	*42, $2A	+43, $2B	,44, $2C	-45, $2D	.46, $2E	/47, $2F
048, $30	149, $31	250, $32	351, $33	452, $34	553, $35	654, $36	755, $37
856, $38	957, $39	:58, $3A	;59, $3B	<60, $3C	=61, $3D	>62, $3E	?63, $3F

Uppercase characters:

@64, $40	A65, $41	B66, $42	C67, $43	D68, $44	E69, $45	F70, $46	G71, $47
H72, $48	I73, $49	J74, $4A	K75, $4B	L76, $4C	M77, $4D	N78, $4E	O79, $4F
P80, $50	Q81, $51	R82, $52	S83, $53	T84, $54	U85, $55	V86, $56	W87, $57
X88, $58	Y89, $59	Z90, $5A	[91, $5B	\92, $5C	]93, $5D	^94, $5E	_95, $5F

Lowercase characters:

`96, $60	a97, $61	b98, $62	c99, $63	d100, $64	e101, $65	f102, $66	g103, $67
h104, $68	i105, $69	j106, $6A	k107, $6B	l108, $6C	m109, $6D	n110, $6E	o111, $6F
p112, $70	q113, $71	r114, $72	s115, $73	t116, $74	u117, $75	v118, $76	w119, $77
x120, $78	y121, $79	Z122, $7A	{123, $7B	|124, $7C	}125, $7D	~126, $7E	**DEL**127, $7F