# SPL Reference Manual
# http://www.clifford.at/spl/

Clifford Wolf

August 8, 2006

# Contents

# Chapter 1

# What is SPL ?

## 1.1 Overview

SPL is an embeddable programming language with a wide range of features:

```
* complete stateful. It is at any point possible to interrupt
  a running SPL script, dump its entire state to disk and resume
  later on.

* feature-rich. SPL has native support for hashes and arrays, regular
  expressions, object oriented programming, etc.

* dynamic. SPL is a fully dynamic language - with all the advantages
  and disadvantages.

* c-style syntax. SPL has a c-style syntax (as well as many other
  languages such as Java, JavaScript, PHP, slang, etc). so it is easier
  to get started.

* advanced string lexing. SPL allows the programmer to simply embed
  variables and complex expressions in strings and template files. E.g.
  this is very important for rapid development of web applications.

* well-structured backend. The SPL runtime is not just one big
  blackbox. Instead, there is a clear and visible seperation of
  compiler, assembler, optimizer, virtual machine, etc. This makes it
  possible to easily adapt the library for your special needs when
  embedding it in your applications.

* more, more, more,..
```

The Name "SPL" is a left-recursive acronym and expands to "SPL Programming Language". The name was meant to be pronounced as an acronym, but I've already heard people pronouncing it "spell", which is also fine with me.

The SPL VM is a pure bytecode interpeter. Support for JIT compilation or generating machine code for the host CPU is not planned and doesn't make much sense for various technical reasons.

The entire SPL toolchain (compiler, assembler, virtual machine, etc) is pretty small (about 100k on x86 architectures). The additional memory usage by the applications is rather small too.

WebSPL is a pretty powerful framework for doing web application development with SPL.

The interesting thing about WebSPL is that, other than usual CGI scripts, a WebSPL script is not executed once for each HTTP request. Instead there is one SPL process for each session and such a WebSPL script can pause it's execution at any time, wait for the user to do something and continue execution as soon as the user did something.

In addition to that there is a module package called "WSF" (WebSPL Forms) which adds an additional abstraction layer between the application logic and the representation as web page. With this SPL modules, web application development becomes as easy as normal application development with well-designed widget sets.

## 1.2 Why should I learn SPL ?

Maybe you want to use some software which is either built with SPL or using SPL as embedded scripting language (such as QCAKE - www.qcake.org). In that case you have answered that question already.

If you are doing Web Application development you defenitely want to have a look at the WebSPL framework (README.WEBSPL or chapter 3 in the manual). It is an entirely new concept of writing web applications.

If you are looking for a scripting language to embed in your applications, SPL might be a great choice. Being easy to embed and to adapt was one of the SPL design goals.

Don't look into SPL if you want to manage a lot of binary data. After all, it is a scripting language.

In all other cases - SPL might still be very interesting for you. It is a very flexible and powerful programming language and maybe you find another interesting field in which SPL is at its full strengths.

## 1.3 Executing SPL scripts on the command line

Often SPL is embedded in other applications. In this cases it depends on the application how the SPL scripts can be executed. But SPL can also be used on the command line using the 'splrun' program. E.g.:

```
splrun hanoi.spl
```

It is also possible to pass a small SPL script directly on the command line:

```
splrun -q 'debug "Hello World!";'
```

SPL is not optimized for fast compilation. Often compilation of a script takes longer than execution. If this is a problem for you, you should pre-compile your SPL scripts to SPL bytecode files:

```
splrun -x hanoi.splb hanoi.spl
```

Such SPL bytecode files can then be executed using the -c option:

```
splrun -c hanoi.splb
```

With the -m option it is even possible to create executeable files with the look and feel of compiled C programs:

```
splrun -m hanoi hanoi.spl
./hanoi
```

Usually it is a good idea to pass the option -e when compiling SPL scripts (or running not pre-compiled scripts). This option enables debug symbols so that runtime error messages are printed with the line number in which the error was triggered. Without this option only the offset in the current bytecode block is contained in runtime error messages.

Calling 'splrun' without any parameter prints an overview of all available command line options.

## 1.4 Little dump/restore demonstration

The 'splrun' runtime and the "termio" module implement some CLIB functions:

```
write(text)     write the text
clear()         clear the screen
setpos(x,y)     set a new cursor position
sleep()         sleep for 1 second
```

These functions are used by the demo 'hanoi.spl' to play a little game of towers of hanoi: type "./splrun -d hanoi.spld hanoi.spl".

If you interrupt the program by pressing Ctrl-C, the full VM state will be stored in the file 'hanoi.spld'. It can be resumed where its has been interrupted any time by calling "./splrun -R hanoi.spld".

## 1.5 Documentation

```
README ........... This short introduction    (manual chapter 1)
INSTALL .......... Installation Instructions  (manual chapter 2)
README.LANG ...... Language Reference Manual  (manual chapter 3)
README.WEBSPL .... WebSPL and WSF Tutorial    (manual chapter 4)
spldoc/ .......... SPL Module Reference       (next manual chapters)
README.API ....... C API Reference            (the last chapter)
```

The module references are built using SPLDOC. Simply run 'make spldoc' to create the "spldoc/" directory with the module documentations.

The SPL Reference Manual is automatically generated from this sources. The PDF can be downloaded from the SPL homepage, a "make spldoc" also creates the "manual.tex" file.

## 1.6 WebSPL

WebSPL makes use of the dump and restore features of SPL and creates a state over the stateless HTTP protocol. Applications can simply generate a webpage and then dump to disk and wait for the user to do something and resume when the user sends a new request.

Install 'webspl.cgi' in your cgi-bin, copy the 'spl_modules/' directory to the same location and create a 'webspl_cache/' directory in cgi-bin as well. This directory must be writable by the apache user, it is used for dump files. Add somthing like this to your apache config (httpd.conf):

```
AddType application/x-httpd-webspl .webspl
Action application/x-httpd-webspl /cgi-bin/webspl.cgi
```

You might also want to restrict access to files such as your templates to prohibit users from simply downloading them as ASCII files.

Note that 'webspl.cgi' needs write-permissions in a subdirectory 'webspl_cache' of the directory which contains 'webspl.cgi'. For just trying out WebSPL, you can also simply start a seperate apache instance in the SPL source directory:

```
httpd -d $PWD -c "DocumentRoot $PWD" -f httpd.conf
mozilla http://localhost:3054/webspl_demo/friends.webspl
mozilla http://localhost:3054/webspl_demo/wsfdemo.webspl
mozilla http://localhost:3054/webspl_demo/wsfgraph.webspl
mozilla http://localhost:3054/webspl_demo/wsf_dialog.webspl
kill $( cat apache.pid )
```

The 'example-apps' package from the SPL webpage contains some additional SPL and WebSPL demo applications. E.g. the 'cfpmanager' and 'splticket' applicationes:

```
mozilla http://localhost:3054/cfpmanager/cfpmanager.webspl
mozilla http://localhost:3054/splticket/splticket.webspl
```

But before you can use them, you need to run the 'setup.sh' script in the 'cfpmanager' or 'splticket' directory respectively.

Instead of apache you can also use the native WebSPL HTTP server. It does not support stuff such as directory indexes and is not optimal for static content, but is really fast for executing WebSPL scripts because it can hold the sessions in memory and only dump them to disk when it is required and it caches the SPL VM bytecode and so avoids redundant recompilations of the same source file:

```
./webspld
```

For best performance it is recommended to use the WebSPL HTTP server for the WebSPL scripts and Apache for the rest. One way of integrating them is using the Apache reverse proxy module.

Both WebSPL runtimes (the cgi script and the HTTP server) are looking for files named 'webspl.conf' in the script directory and all parent directories. This webspl.conf files can be used to configure the WebSPL runtime environemnt and particular WebSPL scripts. These files contain sections, each section starting with '[ filename-pattern ]', containing key-value pairs:

```
[ foo-*.webspl ]
spl.expirelocation = /expired.html

[ bar-*.webspl ]
spl.sessioncookie = bar_sid
spl.respawnsessions = 1
```

The configuration variable 'spl.expirelocation' contains a URL. Browsers will be redirected to that URL when the User is trying to resume an expired session.

The configuration variable 'spl.sessioncookie' contains the name of a cookie which then can be used to pass the session-id (instead of the 'sid' query string variable).

The configuration variable 'spl.respawnsessions', when set to 1, lets the WebSPL framework automatically start a new session when the session passed by the browser has expired.

Any key-value pairs can be specified in the webspl.conf files. Only the few described here are used by the webspl runtime environment, but others can be interpreted by modules or applications (see 'cgi.config' in the module documentation of the "cgi" module).

# Chapter 2

# SPL Installation Instructions

## 2.1 Prerequisites

SPL currently supports Linux (and other POSIX environments), MacOS-X (Darwin), BSD Systems, SGI IRIX, Cygwin (Win32) and native Win32 (using MinGW).

For building SPL you need:

```
- A GNU make (gmake)
- The usual shell scripting tools (bash, awk, etc..)
- An up-to-date GCC (SPL is using some C99 features)
- The Bison parser generator
```

It also is strongly recommendes to have the following packages (including the development files) installed:

```
- The PCRE regular expressions library
- The Readline library
- The pthreads library
```

Without them SPL would be built without some important and often needed features.

SPL comes with a lot of additional modules. Some of them are bindings to external libraries. Theses libraries need to be installed (including the development files) in order to build those additional SPL modules:

```
- The Expat XML SAX parser library
- The SQLite database library (version 3)
- The MySQL database client libraries
- The PostgreSQL database client libraries
- The LibXML2 and LibXSLT libraries
- The SDL and the SDL_Image libraries
- The LibUUID library (from e2fsprogs)
- The SMOKE library (kdebindings)
```

## 2.2 Configuring

Usually it is not needed to configure the SPL build process. It autodetects which libraries are installed and automatically disables some of the features when needed.

The results of this autodetection can be displayed using "make showconfig".

It also is possible to make some manual configurations by editing the "CONFIG" section in the 'GNUmakefile' or altering the 'config.cache' file.

## 2.3 Building and Installing

SPL can be built by running

```
make
```

and installed by running

```
make install
```

On BSD systems one needs to use the program "gmake" instead of "make" for building and installing SPL.

## 2.4 Installing VIM syntax highlighting for SPL scripts

In a final step you might want to install VIM syntax highlighting for SPL scripts. To do so you first need to install the highlighting rules:

```
mkdir -p ~/.vim/syntax
cp syntax-spl.vim ~/.vim/syntax/spl.vim
cp syntax-spltpl.vim ~/.vim/syntax/spltpl.vim
```

And then activate them by adding these lines to your ~/.vimrc file:

```
au BufRead,BufNewFile *.spl,*.webspl set filetype=spl
au BufRead,BufNewFile *.spltpl set filetype=spltpl
```

After that, VIM will automatically put the SPL syntax highlighting rules in place whenever you open an SPL script in the editor.

# Chapter 3

# SPL Language Reference Manual

The SPL sources include many example programs. E.g.:

```
example/example*.spl
hanoi.spl
```

SPL scripts can be executed with "splrun <spl-script-filename>". Web-SPL scripts (*.webspl) can be executed using "webspl.cgi" or "webspld" (see README file or the 1st chapter in the big SPL manual).

## 3.1 Basics

SPL is a C-like language (such as Java or PHP). Constructs such as "if" statements and "while" loops work as in those other languages.

I am not explaining the basic language constructs derived from C in this manual. Please have a look at

```
http://en.wikipedia.org/wiki/C_syntax
```

if you have no idea of how the basic syntax of C (and other C-like languages) looks like.

SPL is a dynamically typed language. So don't wonder how you can define a variable as "int", "float" or "array of objects". Just use the variables as "int", "float" or "array of objects" and they will be.

However - the operators can have types:

```
+ - * / % **
            Dynamically typed addition, subtraction,
            multiplication, division, modulo and power operators.

#+ #- #* #/ #% #**
            Integer addition, subtraction, multiplication,
            division, modulo and power operators.
```

```
.+ .- .* ./ .% .**
            Floating point addition, subtraction, multiplication,
            division, modulo and power operators.

== != <= < > >=
            Dynamically typed comparison operators

#== #!= #<= #< #> #>=
            Integer comparison

.== .!= .<= .< .> .>=
            Floating point comparison

~== ~!= ~<= ~< ~> ~>=
            String comparison

! || &&
            Logical NOT, OR and AND
            (the keywords 'not', 'or' and 'and' are also
            available and do exactly the same thing)

~
            String concatenation

.( ... )
            Cast the expression in the parentheses to
            a floating point value.

#( ... )
            Cast the expression in the parentheses to
            an integer value.
```

The dynamically typed operators are dynamically mapped to the integer, floating point or object operators (the object operators are described in a seperate section below). But they are never mapped to string operators. So if you want to e.g. compare two strings you must always use the '~==' operator.

SPL also has support for the usual combined operation-assign operators such as '+=' (or '~=' for appending to a string). Also the famous '++' and '−−' operators (pre- and postfix) do exist in SPL. The operator '<=>' can be used to switch the values of two variables.

Have a look at the 'hanoi.spl' program in this directory. It is a nice example for a simple SPL program.

## 3.2 Generating Debug Output

The keyword 'debug' may be used to create debug output. The debug message itself is simply passed as argument. Since this is a language keyword (and not a function), no parentheses are required for the argument:

```
debug "Hello World!";
```

The way the debug messages are diplayed depends on the host application embedding the SPL runtime. They may be printed to the console, displayed in popup windows, written to logfiles or simply be ignored.

The keyword 'warning' works the same way as 'debug', but also includes a stack backtrace in the messege and the keyword 'panic' also terminates the program execution.

## 3.3 Variables

As said already, SPL is entirely typeless. There is just the abstract construct of variables. A variable can represent everything, a simple scalar value, an array or hash, a function or an object.

Variables are declared using the "var" keyword:

```
var foobar;
```

Values can be assigned using the '=' operator:

```
foobar = 42;
```

You can also use the variable as array or hash by appending '[ .. ]' to the variable name (as you might already be used to from other languages):

```
foorbar[42] = 23;
```

But more about that in the section about arrays and hashes.

The keyword 'declared' can be used to check if a variable has been declared. The keyword 'defined' can be used to check if a declared variable has a value assigned. The keyword 'undef' represents an undefined value and can e.g. be used to remove the value of a variable:

```
var x;

if (declared x)
      debug "Variable x has been declared.";

if (defined x)
      debug "This is never reached.";

x = 42;

if (defined x)
      debug "Now x is declared and defined.";

x = undef;

if (not defined x)
      debug "Now x is not defined again.";
```

The keyword 'lengthof' can be used to get the length (in characters) of the srtring representation of a variable:

```
debug lengthof "Hello World";
```

Using an undeclared variable (outside a 'declared' test) causes a runtime error.

## 3.4 Functions

Functions can be declared with the "function" keyword, such as:

```
function hypot(x, y)
{
      return ( x.**2 .+ y.**2 ) .** (1./2);
}
```

And may be called by simply using the function name and passing the arguments in parentheses:

```
var h = hypot(5, 12);
```

Note that the parentheses are required when calling functions in SPL. However, there are some language keywords with function-like behavior. But they don't have parentheses around their arguments because they are not functions but compiler keywords:

```
debug, warning, panic, delete, import, load, new, return,
defined, declared, pop, shift, push, unshift, throw
```

Note that functions can be copied as any other variable. So it is e.g. possible to simply pass a callback function as parameter to another function without any syntactical black magic.

It is also possible to define anonymous functions. In this case the function definition evaluates to the function pointer and can e.g. be assigned to a variable. The syntax is the same as for normal function declarations, but the function name is skipped:

```
var hypot = function(x, y) {
      return ( x.**2 .+ y.**2 ) .** (1./2);
};
```

This is especially useful when functions need to be passed as arguments to other functions:

```
array_sort_by_values(myarray, function(a,b) { return a < b; });
```

## 3.5 Array Operations

In SPL, there is no difference between arrays, hashes and other complex data structures like objects. A variable simply may have child variables. The children of a variable may be accessed using the dot-operator or brackets:

```
foo.bar = 42;
foo["bar"] = 42;
```

the only difference between the two methods is that the name of the child variables is limited to the lexical rules for identifiers (the regular expression /[a-zA-Z_][a-zA-Z0-9_]*/) and that child variables must be explicitly defined with the 'var' keyword when accessed using the dot operator.

It is possible to iterate over all children of a variable with the 'foreach' statement:

```
foreach index (array)
      debug array[index];
```

where the 'index' variable is automatically defined by the foreach statement and only valid for the loop body. A foreach loop always returns the elements in the same order in which they have been created (except for elements added using the 'unshift' statement).

The foreach loop usually iterates over the indexes of an array. It is also possible to iterate over the values using the foreach[] loop. So this is identical to the previous example:

```
foreach[] value (array)
      debug value;
```

There are four special variables which only exist within foreach loops:

```
$[ .. is set to '1' when this is the first element and to '0' otherwise
$] .. is set to '1' when this is the last element and to '0' otherwise
$# .. the current index (not the value pointed to by the index)
$@ .. the array which is processed by the foreach loop
```

Arrays may be defined in-place using square brackets. There are two methods of doing so: With automatically assigned and with explicitly declared keys. When declaring the keys, the '=>' operator can be used when the key is specified as value and the ':' operator when the key is specified like a variable name. It is also possible to mix those two methods:

```
var a1 = [ "a", "b", "c" ];

var a2 = [ "x" => "a", "y" => "b", z: "c" ];

var a3 = [ x: "a", 100 => "b", "c" ];
```

has the same effect as:

```
var a1, a2, a3;

a1[0] = "a";
a1[1] = "b";
a1[2] = "c";

a2["x"] = "a";
a2["y"] = "b";
a2["z"] = "c";
```

```
a3["x"] = "a";
a3[100] = "b";
a3[101] = "c";
```

So, when adding elements to an array without explicitly defining a key, the highest numeric key defined in the array will be incremented by one to build the key for the new element.

There are two other instructions which can be used to add elements to an array:

```
push array, 23;
unshift array, 42;
```

The push instruction adds an element to the end of the array, unshift to the beginning. The difference between these instructions is in the position the new element will have in the array, not in the key which is assigned to it. The key is the next integer value in both cases.

There are also instructions for removing the first and last entry from an array:

```
var last = pop array;
var first = shift array;
```

But it is also possible to simply remove elements by using the key:

```
delete array[42];
```

Besides the 'foreach' loop, it is also possible to manually 'walk through' an array using the 'next' and 'prev' instructions:

```
var a = [ 3 => 'x', 5 => 'y', 7 => 'z' ];

var three = next a, undef;
var five  = prev a, 7;
var seven = next a, 5;
```

They return the next or previous index value in the array passed as 1st parameter, relative to the index passed as 2nd parameter. If the 2nd parameter is undef, 'next' returns the first and 'prev' the last index. If there is no next or previous element, undef is returned.

The keyword 'elementsof' can be used to get the number of elements in an array:

```
debug elementsof [ 1, 2, 3, 4, 5, 6 ];
```

Additional functions for working with arrays (such as sorting) are provided by the "array" module which is described in SPLDOC (there are separate sections about SPL Modules and SPLDOC later in this document).

## 3.6 Vaargs like arguments

SPL also supports 'vaargs' like argument passing. The last argument in a function declaration may be prefixed with an '@' character. In that case the variable representing this argument will be an array of all remaining arguments. If an '@' character is put in front of a function argument when calling a function, that argument is interpreted as an array and an argument for each array element will be inserted. E.g.:

```
function foo(@args) {
      foreach i (args)
            debug args[i];
}

function bar(@args) {
      foo("one", @args, "four");
}

bar("two", "three");
```

This is useful for writing wrapper functions and functions with a variable number of arguments.

## 3.7 Named Function Arguments

SPL also supports named arguments (aka options). With this arguments, not the position in the function call but the assigned name identifies an argument. All named parameters are assigned to a hash which is declared with a '%' prefix in the function prototype. The '%' prefix may also be used when calling a function to pass all elements of a hash as named arguments. E.g.:

```
function foo(%args) {
      foreach i (args)
            debug "$i -> ${args[i]}";
}

function bar(%args) {
      foo(one: 1, %args, four: 4);
}

bar(two: 2, three: 3);
```

This is e.g. useful for function arguments which are rarely used.

When a named option is specified multiple times, the most left specification is used. That way it is possible to pass a hash with default values as last parameter using the '%' prefix.

## 3.8 Objects

If you have no idea what object oriented programming is, read

```
http://en.wikipedia.org/wiki/Object_Oriented_Programming
```

first.

SPL does not really know about the difference between classes and objects. If only one instance of a class/object is needed, it is possible to simply use the name defined with the 'object' keyword. (But if you do so, no constructor will be executed.) Because of this, we don't speak about classes and objects in SPL, but of objects and instances of objects.

Objects can be defined in SPL like this:

```
object Foo
{
      var counter;

      method increment_counter()
      {
            return counter++;
      {

      method init(start_value)
      {
            counter = start_value;
            return this;
      }
}
```

The method 'init' has a special meaning: It is the constructor. There are no destructors in SPL, but it is common to create a method named 'destroy' for the job and call it manually when the object isn't needed anymore.

The constructor must return a self pointer. The keyword 'this' can be used to create such a pointer. Thus the "return this;" in the example above.

If the new object is derived from another one, the parent object is specified right after the object name in the object definition:

```
object Bar Foo
{
      method decrement_counter()
      {
            return counter--;
      }
}
```

Objects may be instantiated using the 'new' operator. If any arguments are specified when instantiating an object, they are passed through to the object constructor as they are:

```
var mycounter = new Foo(42);
```

SPL does support nested functions as well as nested objects. So it is possible to define 'local objects' in your functions.

It is also possible to derive one object from more than one parent. This can be done using the 'import' statement:

```
object Foobar Foo
{
      import Bar;

      [...]
}
```

But the parent objects which are 'imported' neither show up in the object derivation path of the object, nor in its reflection string.

When an object or one of its instances is used as scalar variable, they evaluate to their reflection strings. E.g.:

```
object A { }
object B A { }
var x = new B();

debug A; // prints: SPL Debug: A
debug B; // prints: SPL Debug: A | B
debug x; // prints: SPL Debug: [ A | B ]
```

Objects and instances are automatically created as references. So they can be passed as arguments and being returned from functions without implicitly being copied by those operations.

If a method is overloaded in a child object and the implementation from the parent object needs to be called in the child object, this can be done using the context dispatching operator '*':

```
object A {
      method foo() {
            debug "Now in foo from A.";
      }
}

object B A {
      method foo() {
            debug "Now in foo from B.";
            *A.foo();
      }
}
```

Without the '*', A.foo() would be called in the context of A and not in the context of B (or the currently active instance of B).

The keyword 'static' can be used to declare static object variables. A static object variable is shared between all objects in the derivation path and all instances of these objects. The declaration works exactly the same way as for declaring normal variables in objects, just that the keyword 'static' is used instead of the keyword 'var'.

There is nothing like 'private functions' in SPL. All methods (and variables) are public. If some are intended for internal use only, they must be protected by choosing a name which avoids collisions. There is a separate section on the recommended naming conventions later in this document.

## 3.9 Object Operators

There is no operator overloading support in SPL. Instead, there are object operators (as they are seperate integer and floating point operators too). Whenever such an object operator is used, an "operator_*" method is called in the left operand and both operands are passed as parameters. The method name is different for each object operator:

```
(+)      and      (+)=      are calling the method operator_add(a, b)
(-)      and      (-)=      are calling the method operator_sub(a, b)
(*)      and      (*)=      are calling the method operator_mul(a, b)
(/)      and      (/)=      are calling the method operator_div(a, b)
(%)      and      (%)=      are calling the method operator_mod(a, b)
(**)      and      (**)=      are calling the method operator_pow(a, b)

(<)      is calling the method operator_lt(a, b)
(>)      is calling the method operator_gt(a, b)
(<=)      is calling the method operator_le(a, b)
(>=)      is calling the method operator_ge(a, b)
(==)      is calling the method operator_eq(a, b)
(!=)      is calling the method operator_ne(a, b)
```

The operator precedence is the same as for the normal integer and floating point operators. The generic dynamically typed operators (+, -, *, etc.) are automatically mapped to the object operators when used with objects.

## 3.10 Exceptions

SPL has an exception handling mechanism for error-reporting and -handling. The exceptions themselves are objects. In fact, every object could be used as an exception. But it is recommended to only use objects as exceptions which have been designed for this purpose. Usually the names of such exception objects end with the "Ex" postfix. E.g. all exceptions thrown by the SQL modules are of the type "SqlEx".

Exceptions may be thrown using the keyword 'throw' and are catched in 'try' blocks. Here is a small example:

```
object MyEx { }
object MySpecialEx MyEx { }

try (x)
{
      throw new MySpecialEx();
      debug "** this line is never reached **";

catch MyEx:
      debug "Got a 'MyEx' exception. The backtrace is:\n" ~
            x.backtrace;
}
```

If no 'catch' rule matches the exception object or any of its parent objects, a runtime error is printed and the execution of the program is terminated.

The 'throw' instruction automatically adds the 'backtrace' variable to the exception object. This variable contains a human-readable backtrace of the current task. If the exception object has a 'description' variable, its text value will be printed as part of the runtime error for uncatched exceptions.

It is considered good coding style to only use exceptions for error handling.

## 3.11 Strings and Here-Documents

Strings can be quoted using double (") or single (') quotes. The only difference between the two quoting styles is that within double quotes it is possible to use single quotes unescaped and vice versa.

Strings can be concatenated using the ~ operator. But a list of string constants is concatenated automatically. So this three code snippets are identical:

```
debug "hello world";
debug "hello" ~ " " ~ "world";
debug "hello" " " "world";
```

The usual backslash escape sequences (such as '\n' for newlines) can be used in both quoting styles. Additionally, there are some special substitution sequences which are all starting with a dollar sign ($). These sequences are described in the next section. The section after it describes so-called templates, an even more advanced way of handling strings in SPL programs.

Another way of quoting strings in SPL are so-called Here-Documents. They work a little bit differently in SPL compared to other languages such as Perl.

An example:

```
debug <<EOT
This is a debug message.
It is quoted using the Here-Documents mechanism.
Substitutions work here: 5 + 3 = ${ 5 + 3 }
EOT;
```

The Here-Document starts with '<<Token' and is terminated by the next occurrence of 'Token'. A white space character (i.e. also a newline character) after '<<Token' is ignored. Unlike in other languages, the terminating 'Token' doesn't need to be at the beginning of a new line. Also note that the ';', which terminates the command, is located after the end token. The Here-Document starts immediately after '<<Token'. The special substitutions (which are described in the next section) do work in this kind of Here-Document.

If '>>' instead of '<<' is used, the Here-Document is literal and no substitutions are performed.

The '<<Token' (or '>>Token') may be followed by any special character, which is then used as so-called indenting character. All characters from the beginning of a line in the Here-Document until the first occurrence of that indenting character are ignored:

```
debug <<EOT:
      :This Here-Document is using indenting characters.
      :So there is no indenting in the output.
EOT;
```

In addition to these types of Here-Documents, a '<<<' marks the begin of a Here-Document until the end of the line, with substitutions enabled. A '>>>' does the same, but without processing any special substitutions.

## 3.12 Special Substitutions in Strings

SPL has support for some special substitution operators in strings and templates (the latter are described in the next section). All these substitution operations start with the dollar sign ($):

```
$variable

    Will be substituted by the current value of the variable.

${expression}

    Will be substituted by the value of the expression. This can
    be everything - including a really complex program code snippet.

$(function)

    This is an embedded function. It will be executed and the return
    value substituted. E.g.:

          $( if (should_insert_foo()) return "Foo"; return "Bar"; )

$<Name>

      Insert results from a regular expression (by name or numeric ID). The
    brackets can be left away for single digit IDs, $- and $+.

$[ comment ]

    This is a comment. It is always substituted with an empty string.

$$, $:, $?

    Just a "$", ":" or "?" respectively.
    (for escaping $.., <spl:.. and <?spl...)

$ whitespace sequence

    Don't include the whitespace sequence in the output.

$ at end of line
```

```
    Don't include the line break in the output.
```

Use complex substitutions with care! Substitutions are powerful tools, but when they are too excessively used you get programs no one will ever understand again.

## 3.13 Templates

Templates are another form of Here Documents. They start with <Token> and are terminated by </Token>. It is possible to use an empty string as token too, so <> ... </> also works fine. Within templates, all the $ substitutions are supported. In addition to them, there are also some special XML-like tags allowed in templates:

```
<spl:comment>
...
</spl:comment>
    A comment. Everything between the tags will be ignored.

<spl:if code="SPL Code">
...
</spl:if>
    The SPL expression passed using the "code" attribute will be
    evaluated and the content of the tag will only be embedded if the
    result is true.

<spl:else>
...
</spl:else>
    This tag must directly follow a <spl:if> tag. Its content will
    only be embedded if the content of the <spl:if> tag has not been
    embedded.

<spl:code>
...
</spl:code>
    The content of this tag will be evaluated as embedded function.
    The return value (if any) is substituted. (like "$( ... )")

<spl:code code="SPL Code">
...
</spl:code>
    The same as above, but the SPL program code is passed using the
    "code" attribute. The content of the tag is available through a
    variable with the name "_data_" when executing the code. This is e.g.
    useful if you want to pass a part of your document through an encoding
    function:

          <spl:code code="xml::_data_">
```

```
            This is xml encoded: < & >
        </spl:code>
```

The encoding operator (::) is described later.

```
<spl:inline>
...
</spl:inline>
```

The <spl:code> tag is evaluated like an embedded function. So e.g. variables declared in an <spl:code> tag is local to that tag. The <spl:inline> tag is evaluated in the same context as the template text itself. So it can be used to e.g. declare functions or variables used in the rest of the template. Do not use the 'return' keyword in an <spl:inline> tag unless you really want to return from the the current function context. A <spl:inline> tag is always substituted with an empty string.

```
<spl:var var="Variable Name">
...
</spl:var>
```

This tag is always substituted with an empty string, but the variable named in the "var" attribute will be declared and set to the content of the tag.

```
<spl:var var="Variable Name" code="SPL Code">
...
</spl:var>
```

As above, but the value will again be passed through the specified SPL expression and is available as "_data_" in the expression.

```
<spl:foreach var="Iterator" list="List">
...
</spl:foreach>
```

Builds a foreach instruction using the variable name passed in the "var" attribute as iterator and the list passed in the "list" attribute as list to iterate over. If the behavior of a foreach[] loop is wanted, the variable name in the "var" attribute must be prefixed with "[]".

```
<spl:_>
...
</spl:_>
```

Localize the tag body (see "Internationalization and Localization" below). Localization domains can be passed by using the syntax <spl:_domain_>...</spl:_domain_>. Note that no other special tags are allowed inside of this tag.

```
<spl:indent char="IndentingCharacter">
...
</spl:indent>
```

The tag body is parsed using the specified indenting character.

Here is a small example of what can be done using SPL Templates. It selects messages from a database and creates HTML code for showing them in java script popup windows.

```
<spl:var var="tuples" code="sql(db_handler, _data_)">
      SELECT message, timestamp
        FROM message_table
       WHERE user_id = ${sql::userid}
   <spl:if code="clock_hours < 8 || clock_hours > 18">
         AND urgent = 1
   </spl:if>
</spl:var>

<spl:foreach var="i" list="tuples">
    <script><!--
      alert(
          '<spl:code code="js::_data_">
            At ${tuples[i].timestamp}: ${tuples[i].message}
          </spl:code>'
            );
    //--></script>
</spl:foreach>
```

SPL also has support for PHP-like code snippets in templates using '<?spl' .. '?>' tags. The main difference to the <spl:code> and <spl:inline> tags is that it is possible to let SPL code blocks and template blocks overlap with this syntax. E.g.:

```
First Line
<?spl
      var list = [ "Hello", "World!" ];
      foreach i (list) {
            var word = list[i];
?>
            word #$i is '$word'
<?spl
      }
?>
Last Line
```

The <spl:code> and <spl:inline> tags would require the '{' .. '}' block to be closed in the same code block in which it has been opened.

Templates can also be started with <:Token>, in which case the ':' is used as indenting character. This does not change the termination string for the token (it is still </Token>). Other indenting characters than ':' are not supported for inline templates (except they are introduced using the <spl:indent> tag).

As with the substitutions, you should consider wisely how to use SPL Templates in your programs because it is also possible to use them to produce obfuscated code.

## 3.14 The <splif:*> and <splcall:*> Template Tags

In addition to the <spl:*> template tags described above there is also support for <splif:*> and <splcall:*> template tags.

The <splcall:*> tags are transformed to calls to splcall_*() functions. The tag body is transformed to a function which evaluates it and this function is passed as argument to the splcall_*() function. The return value of the splcall_*() is substituted. Example given:

```
function splcall_toupper(textfunc) {
      var text = textfunc();
      text =~ e/[a-z]/g chr(ord($0)+ord('A')-ord('a'));
      return text;
}

debug <>This is <splcall:toupper>very important</splcall:toupper>!</>;
```

The tag attributes are passed as named parameters to the splcall_*() functions. When the attributes are quoted with ' or " they are interpreted as strings and when they are quoted with ( .. ) they are interpreted as SPL expressions. It is also possible to ommit the tag body by terminating the tag with '/>'. Example given:

```
function splcall_getuser(%args) {
      return sql_value(db, "SELECT username "
                  "FROM user WHERE id = $args.id");
}

debug <>User $userid: <splcall:getuser id=(userid)/></>;
```

The <splif:*> tags work exactly like the <splcall:*> tags, but instead of passing the tag body as argument and substituting the return value the tag body is only evaluated and included in the template output when the return value of the splif_*() function returns true. It is not possible to ommit the tag body of <splif:*> tag. The <spl:else> tag can also be used with an <splif:*> tag. Example given:

```
function splif_adminuser(%args) {
      return sql_value(db, "SELECT isadmin "
                  "FROM user WHERE id = $args.id");
}

debug <:>
      : Data: $data_string
      <splif:adminuser id="$userid">
      : Secret Data: $secret_data_string
      </splif:adminuser>
      <spl:else>
      : Secret Data: ** admins only **
      </spl:else>
</>;
```

The <splif:*> and <splcall:*> tags don't really add any functionality which are not already provided by the <spl:*> tags. But they can be used to simplify writing SPL template files a lot, especially if the person writing the template files is not a programmer.

## 3.15 Loadable Modules

There are two different types of loadable modules: SPL byte code modules and machine byte code modules. Machine byte code modules are *.so files (or *.dll on Win32 host). The README.API file (and the last chapter in the manual) describes how to write such modules in C. SPL byte code modules are written in SPL and compiled using the "splrun" command line tool.

Both types of modules are loaded using the 'load' instruction. E.g.:

```
load "sql";
```

When executing such an instruction, the SPL virtual machine is first looking for a machine byte code module with such a name in the module search path and then for SPL byte code modules.

The module files are named "mod_<Module-Name>.splb" (or "mod_<Module-Name>.so" or "mod_<Module-Name>.dll" respectively).

Each module can only be loaded once. Additional 'load' instructions for an already loaded module are ignored.

A module is written like any other SPL program, but should only declare variables, functions and objects and should not execute any real code (or at least limit that to some simple initializations).

SPL source files can be compiled to byte code files like this:

```
./splrun -N -e -x mod_foobar.splb mod_foobar.spl
```

The -e is optional and instructs the compiler to include debug symbols in the byte code file. Running 'splrun' without any parameters prints out the full list of available command line options.

## 3.16 SPLDOC Comments

Modules should have SPLDOC Comments in their source, to make it possible to create API references for them directly from the source code with the 'spldoc.spl' tool.

Running 'make spldoc' in the SPL source dir creates the documentation to all modules which are included in the SPL distribution. The generated API references are written to the "spldoc/" directory. They can also be found on the SPL web page.

SPLDOC comments look like this:

```
/**
 * This is an SPLDOC comment for the foobar function
 */
function foobar();
```

So they start with "/**" on a line on its own. Then comes the description text. If the first character in the line is an asterisk (*), it will be ignored. The SPLDOC comment ends with '*/' and right in the next line (a blank line is not allowed here) comes the commented function prototype (or variable declaration, etc).

The first SPLDOC comment in a module source file is special: it describes the module itself (so the first line after the comment has no special meaning in this case).

In order to be parse-able by SPLDOC, a source file must be "well formatted". That means, the functions, variables, etc. which are not an object member must be declared (and commented) first, then the objects. It is important that not only the object members, but also the object itself is commented. Otherwise SPLDOC wouldn't 'see' the object and the documentation wouldn't match the module. In an object, you should always define static variables first, then dynamic variables, then functions and then methods. Everything which is not an object member must be defined before the first object and should be defined in the same order (first variables, then functions).

Have a look at "spl_modules/mod_wsf.spl" and "spldoc/wsf.html" (which is created by 'make spldoc') for a good example.

## 3.17 Naming Conventions

SPL has a flat root name space. So it is important to have some naming conventions to avoid collisions in this flat name space.

The module names themselves are using a kind of hierarchy. E.g. there is a module called "wsf" and a module "wsf_dialog" based on it. In the module "wsf" all object names start with the prefix "Wsf" and all functions and global variables start with "wsf_". In the "wsf_dialog" module, the object prefix is "WsfDialog" and the function/variable prefix is "wsf_dialog_".

A few more words about object names. Each part of the object which start a new 'logical block' (and not each word!) should start with a capital letter. E.g.:

```
object Myobject { }
object MyobjectFoobar Myobject { }
```

or:

```
object MyobjectBase { }
object MyobjectFoobar MyobjectBase { }
```

But _not_:

```
object MyObject { }
object MyObjectFooBar MyObject { }
```

It's up to you to decide what a "logical block" is in your case. There are some additional naming conventions for exception objects: They all end with "Ex", but if e.g. exceptions are derived from each other (and some text is appended to the object name), it is appended before the "Ex" postfix.

Private variables and functions of modules (which are not documented using SPLDOC and are for internal use in the module only) must begin with "__<module-name>". Private methods of an object (which are not documented and should not be called from methods in derived objects) must begin with "__<object-name>".

## 3.18 Regular Expressions

If you have no idea what regular expressions are, read

```
http://en.wikipedia.org/wiki/Regular_expressions
```

first.

SPL is using the PCRE library for regular expression matching. So it is pretty compatible with Perl regular expression. If the PCRE library cannot be found by the SPL GNUmakefile, SPL is compiled without regular expression support and a runtime error is produced whenever regular expression instructions are executed.

The syntax for regular expression matching is similar to the Perl syntax:

```
x =~ /foobar/;
x =~ s/foo/bar/g;
```

Perl-like modifiers supported by SPL:

```
i .. ignore case in pattern matching
s .. dot metacharacters also match the newline charater
x .. ignore unescaped white spaces and allow comments using '#'
m .. multi line matching, ^ and $ also match newline characters
g .. match (and substitute) globally, not only the first match
```

Modifiers new in SPL:

```
N .. include captured strings as child nodes in result, using numbers
P .. include named captured strings (?P<foo>...) in result, using names
A .. add an array with an element per match (with 'g' modifier)
R .. when substituting, return new text and keep original unmodified
I .. declare named captured strings (?P<foo>...) as local variables
E .. store the text between the matches in $- (before) and $+ (after)
L .. the $-variables have the data from the last match (with 'g' mod.)
S .. return the text fragments between the matches (split mode)
```

The return value of '=~' is the number of matches found (except when the 'R' modifier is used). If the 'g' modifier isn't used, the return value may only be 0 or 1. With the modifiers 'N', 'P' and 'A', the result will also have child variables with additional data about the matches.

When the 'A' modifier is used without 'N' or 'P' the result is an array with the entire matches. With the 'N' or 'P' modifier the result is an array with the captures as child variables. So splitting up an input file in lines can be done like this (the file_read() function is provided by the "file" module):

```
var lines = file_read("demo.txt") =~ /[^\n]*/Ag;
```

It is possible to declare names for capturing parentheses using the python syntax (?P<name>...). This is much of a help when dealing with complex regular expression with many capturing parentheses.

Referring to the strings matched by a regular expression can be done by using $N, $<Number> and $<Name> (in addition to including them in the result value using the 'N', 'P' and 'A' modifiers). The special variable $0 represents the whole matched text and is also available if no capturing parentheses were present in the regular expression. These special variables are declared locally - they do not invalidate regular expression results in any higher context. So it is save to e.g. do a regular expression, then call a function which is also using regular expression, and after that refer to the matches of the first regular expression using these special variables.

The special modifier 'E' stores the text which has not matched in $- (before the match) and $+ (after the match). In combination with 'N' or 'I' there will also be ["-"] and ["+"] elements in the returned object. When the 'E' modifier is used together with 'g', $- and $+ contains the text snippets between the matches.

A very special variable is the variable $$. It is the local variable in which the regular expression results are stored. So instead of writing $0 you could write $$[0], or $$.foobar instead of $<foobar>. This variable is function local, but it is possible to write 'var $$;' in a local block to create a seperate set of regex result variables. It also is possible to backup the value of $$ in another variable and restore it later on.

Here is a nice example for using complex regular expressions:

```
var x = "foolish bigfoot";

var r = x =~ /(?P<word>(?P<firstchar>\S)\S*)\s*/ALPg;

foreach i (r) {
      var $$;
      r[i].word =~ s/foo(.*)/bar$1/;
      debug "Match #$i: [${r[i].firstchar}] ${r[i].word} ($0)";
}

var text1 = "Ever seen a ${r[0].word} $1?";
var text2 = text1 =~ s/seen/beeing eaten by/R;

debug text1;
debug text2;
```

This script creates the following output:

```
SPL Debug: Match #0: [f] barlish (foolish)
SPL Debug: Match #1: [b] bigbart (foot)
SPL Debug: Ever seen a barlish bigfoot?
SPL Debug: Ever beeing eaten by a barlish bigfoot?
```

It is possible to use the import statement with $$. That has almost the same effect as using the 'I' modifier, but gives you a better control on where the variables for the regular expression matches are declared:

```
var text = "Hello World";

if (text =~ /(?P<foo>\S+)\s+(?P<bar>\S+)/) {
      import $$;
      debug "$foo $bar";
}
if (declared foo)
      panic "This is never reached";

if (text =~ /(?P<foo>\S+)\s+(?P<bar>\S+)/I) {
      debug "$foo $bar";
}
if (declared foo)
      debug "Foo is now defined here too.";
```

A full description of the regular expression syntax supported by PCRE (and SPL) can be found in the "pcrepattern" man page.

Instead of the slash (/) as quoting character for regular expressions, it is also possible to use colons (:), commas (,), exclamation marks (!), percentage sign (%) and the at-character (@).

It is also possible to use regular expression substitutions in which the substitution text is re-evaluated for every match. But the syntax for this is slightly different as in perl:

```
var text = "Some ASCII Codes: A = #A, B = #B, C = #C, D = #D";
debug text =~ e/#(.)/Rg ord($1);
```

For complex to-be-evaluated expressions it is required to put the expression in parentheses. Note that not all of the modifiers listed above are allowed for and that the $-variables are not set by e// expressions.

## 3.19 References

Handling references (aka pointers) is easy in SPL. It is done more or less automatically by the virtual machine: Simple variables (such as scalars or functions) are always passed by value and complex variables (arrays, objects and everything else with child variables) are passed by reference.

At least that's what it looks like. In fact the SPL virtual machine implements a complex copy-on-write behavior, but this is hidden in the machine internals.

The basic idea behind copy-on-write systems is discussed at

```
http://en.wikipedia.org/wiki/Copy_on_write
```

Sometimes it is neccessary to create recursive copies of complex data structures. This can be done by assigning the variables using the ':=' operator (instead of using the normal '=' operator).

It is possible to test if two variable names point to the same object using the '^==' operator. Testing for not-equal is performed with the '^!=' operator.

## 3.20 The Quoting/Encoding Operator (::)

There is a special operator in SPL for quoting and encoding text (::). In fact it is nothing else than a simple function call:

```
foo::bar
```

is identical to

```
encode_foo(bar)
```

but in some cases easier to write and read. There are various modules which provide encode_* functions. E.g. the "sql" module provides an "encode_sql()".

## 3.21 Embedded Functions

Embedded functions are functions which are simply "inlined" in an expression using the special parentheses "({" ... "})".

E.g. this code fragment:

```
var x = 42;

function foobar() {
      if (x == 42) return 23;
      if (x == 23) return 42;
      return 0;
}

debug foobar() + 23;
```

does the same thing as this code fragment:

```
var x = 42;

debug ({
      if (x == 42) return 23;
      if (x == 23) return 42;
      return 0;
      }) + 23;
```

The mechanism used here is the same as the one used for the $( ... ) substitution described above.

## 3.22 Gotos, break and continue

SPL has support for gotos. The syntax is pretty much the same as in C:

```
var i;
for (i=0; i<42; i++) {
      if (i == 23) goto break_out;
jump_back:;
}

if (i > 42)
      return;

break_out:
write("demo2: Now i is $i.\n");
goto jump_back;
```

A goto label must always point to an instruction (as in ANSI C). That's why there is a ';' after the "jump_back" label: The ';' adds an empty instruction.

It is not possible to use gotos to jump from one function in another (or from a function to the main program).

Also note that statements such as 'var' are compiled to real virtual machine statements and if a goto is used to jump over a 'var' statement, the variable won't be declared and any use of that variable would result in a runtime error. So you should only use gotos to do stuff such as jumping out of loops.

SPL also has support for the 'break' and 'continue' statements. They are internally implemented as gotos.

## 3.23 Switch Statements

The switch statements are different in SPL compared to languages such as C. An SPL switch statement actually is nothing else then a series of "if .. else if" statements with a different syntax:

```
var list = [ 1, 2, 3, 4 ];

while (1)
      switch {
            var x = shift list;
      case x == 1:
            debug "x is 1";
      case x == 2:
            debug "x is 2";
      case x == 3:
            debug "x is 3";
      default:
            debug "whatever!";
            exit;
      }
```

The code before the first 'case' statement is always executed. This code block can be used to declare variables which are only used in the switch block. Each case block comes with a condition which defines if that block should be executed. The first case block with a true condition will be executed.

Note that the SPL runtime does not optimize switch statements (e.g. using lookup tables). So a huge list of cases should better be implemented using a hash with function references.

## 3.24 Compiler Pragmas and Preprocessor Statements

There are some compiler pragmas and preprocessor statements for the SPL compiler. First of all there are three statements for including external files at compile time.

The compiler must be able (allowed) to read files when the pragmas for processing files are executed. This is for example not the case for the small code snippets which are compiled by the various eval implementations provided by some modules.

The file include statements are:

```
#file-as-const Filename
#file-as-code Filename
#file-as-template Filename
#file-as-bytecode Filename
```

All four statements include an external file. The first includes a file as string constant with no additional processing. The 2nd just continues compilation in the specified file (and returns when the end-of-file is reached). The third includes the file as template. This is pretty similar to #file-as-const, but $-substitutions and <spl:...> tags are evaluated (see "Templates" above). The fourth includes (aka. 'links in') an already precompiled SPL bytecode file.

If the filename is prefixed with an asterisk character (*), the file is interpreted as so-called embedded-file. Embedded files must be declared in the same SPL program file as they are referred to and are similar to the Perl __DATA__ construct.

Embedded files are declared (a little bit like Here-Documents) using:

```
#embedded-file Filename Token
....
Token
```

The declaration can be anywhere in the file, but usually it is done after the actual program code.

A very different kind of compiler pragma is '#encoding'. SPL usually expects all input in UTF-8. But if your files are not UTF-8 encoded, the '#encoding' pragma can be used to specify the encoding:

```
#encoding iso8859_1
```

At the moment only the encodings "ascii", "iso8859_1" and "latin_1" (these are three names for the same character set) and "utf_8" are known to the SPL compiler.

SPL also has statements for defining and deleting macros:

```
#define pi 3.14159265
#define mysqrt(x) ((x)*(x))

debug sqrt(pi);

#undef pi
#undef mysqrt
```

The macro value is terminated by the end of the line. Multi-line values are also possible by beginning all additional lines with a backslash character.

## 3.25 Hexadecimal, Octal and Binary Numbers

Hexadecimal, octal and binary numbers can be used in SPL with the prefixes '0x', '0o' and '0b'. So the following 4 lines of code are equal:

```
debug 255;
debug 0xff;
debug 0o377;
debug 0b11111111;
```

The C-like prefix '0' for oktal numbers does not work here. Numbers with leading zeros are interpreted as decimal numbers in SPL.

## 3.26 The 'eval' statement

The 'eval' statement can be used to to execute dynamically created SPL code. Example given:

```
eval "debug 'Hello World';";
```

An 'eval' returns -1 on compiler errors and zero otherwise. It is strongly recommended to check the return code of an eval statement.

## 3.27 Hosted Variables

Hosted variables (hnodes) are variables which are managed by SPL modules. Usually they are handlers such as open database connections. The module documentations describe the behavior of the hosted variables provided by the modules.

Such hosted variables usually behave more or less like they are normal SPL variables. But some of them are very different. Some operators - especially those used for array and object operations - may show a very differnt bahvior. So read the module documentations carefully and don't wonder if e.g. such a variable looks like an assoziative array but the foreach loop doesn't work with it.

## 3.28 Internationalization and Localization

The SPL builtins library provides some bindings for the standard gettext setlocale(), bindtextdomain(), textdomain(), gettext() and dgettext() functions.

In addition to that there also exists a special localization operator (the underscore) which can be used as prefix for string constants in SPL:

```
debug _"Hello World!";

debug _<:>
  : The underscore can also be used as prefix for inline templates.
  : But that disables the support for the <spl:*> tags.
</>;
```

The special thing about that operator is that the dollar substitutions are handled a different way in such strings. Example given the statement

```
debug _"The sum of $a and $b is ${a+b}.";
```

Is automatically transformed by the SPL compiler to a call to a special translation function:

```
debug _("The sum of {0} and {1} is {2}.", undef, a, b, a+b);
```

(The 2nd parameter is the text domain to be used. Passing undef means that the text domain set using textdomain() should be used.)

That way it is possible to even translate messages that contain substitutions by creating a .po file containing something like:

```
msgid "The sum of {0} and {1} is {2}."
msgstr "Die summe von {0} und {1} ist {2}."
```

When a different text domain than the one set by the last call to textdomain() should be used, the prefix '_DOMAINNAME_' can be used instead of a simple '_'. For example when the above message should be translated using the 'foobar' domain, the following code could be used:

```
debug _foobar_"The sum of $a and $b is ${a+b}.";
```

A dummy C file with all the translatable strings in a SPL source file can be generated using 'splrun -NX'. This dummy C file can then be used as input for the xgettext program for creating or updating .po files:

```
splrun -NX demo.spl | xgettext -C -
```

A more detailed description of the generic gettext API and the tools can be found in the 'gettext' info page.

## 3.29 Walking through Contexts

First of all: This section is more meant as information about what is possible than a recommendation to actually use the methods described here. SPL variables (aka SPL "nodes") may have child nodes. Those child nodes have unique names and a defined order. The arrays described above are in fact just nodes with such child nodes. When addressing nodes in that tree, i.e. when creating a path in that node tree, a dot is used to tell the virtual machine which part of the path specifies the parent, which the child, grandchild, etc. If parts of the path need to be created dynamically, the [ .. ] operator is used. E.g.:

```
foo.bar[1234]
```

will instruct the SPL virtual machine to look up the variable "foo", then look up its child "bar" and then its child "1234".

But where to look for "foo" in the first place? Each task has a so-called context node. That is the node in which all the local variables of the current command block are stored (as children). If the variable can't be found in the current context, it will be looked for in the context node of that context node, and so on.

That means, whenever a new command block is opened (e.g. with '{'), a new context node is created and the old task context node becomes the context node of the new task context node. The '}' destroys the new context node and the old context node becomes the new task context node again. (The foreach and for loops create a local context even if the loop body is not a '{' ... '}' block.)

All this is done automatically by SPL and the resulting behavior is exactly as one would expect it from a C-like programming language. But here comes the interesting part:

If a variable name starts with '[*]', it means that one context is skipped in the look up path. So it is possible to address the upper contexts directly. It is for example possible to declare global variables from a function context:

```
function create_foobar(value)
{
      var [*].foobar = value;
}

create_foobar(42);
debug foobar;
```

It is even possible to write to the context pointer directly and so change the context pointer. E.g.:

```
object A {
      var foobar = "I am 'foobar' from object A.";
}
object B {
      method print_foobar() {
            if (declared foobar)
                  debug foobar;
```

```
            else
                  debug "Variable 'foobar' not found.";
      }
}

B.print_foobar();

B.[*] = A;
B.print_foobar();
```

The '[*]' operator skips over local command blocks and only stops at function (or other non-local) contexts.

One possible use case of all this is inserting a lookup context in splcall_*() functions. That way it is possible to implement loops with <splcall:*> tags which declare their own local variables. But other that with the context dispatching operator (the '*' prefix for function calls) the <splcall:*> body can still access the local variables from the context in which the body has been declared. Example given:

```
function splcall_myloop(textfunc, %args)
{
      var mytextfunc, myctx, text;

      mytextfunc := textfunc;
      myctx.[*] = mytextfunc.[*];
      mytextfunc.[*] = myctx;

      for (var i = args.from; i<=args.to; i++) {
            myctx["counter"] = i;
            text ~= mytextfunc();
      }

      return text;
}

write(<:>
      <spl:foreach var="[]i" list="['A', 'B']">
        <splcall:myloop from="7" to="9">
            : $i: Current counter: $counter
        </splcall:myloop>
      </spl:foreach>
</:>);
```

In addition the to '[*]' operator there is also the '[+]' operator for following the class pointer (pointing to the parent of an object), the '[/]' operator always points to the root node and the '[.]' operator always points to the current object (aka 'this' pointer).

## 3.30 Command Blocks without local context

Command blocks declared with '{' ... '}' have their own local context. Thus, variables or functions declared in that block are only local to this context. It is also possible to declare command blocks without a local context using the '{[' ... ']}' brackets. Example given:

```
var msg = "";
function x() { msg ~= "World"; }

/* append "Hello " */
{ function x() { msg ~= "Hello "; } x(); }

/* append "World" */
x();

/* append "!" */
{[ function x() { msg ~= "!"; } ]}
x();

debug msg;
```

This program prints "Hello World!". It wouldn't if '{' ... '}' would not have a local context or '{[' ... ']}' would have one.

## 3.31 Inline Assembly

This is another thing which is more of academic interest. Since the SPL compiler creates byte code for the SPL virtual machine, it is possible to create code for that machine more directly too. One way of doing that is by using SPL assembler code. The compiler allows inlining assembler code.

The keyword "asm" expects a list of assembler statements. Each statement in an extra string constant (no substitutions, etc are allowed in these strings). There is no delimiter for these string constants. E.g.:

```
asm 'pushc "Hello World"' 'debug';
```

There isn't anything which can be done by using the assembler but can't be done by the high-level language. So the inline assembler is more important for academic purposes or obscure optimizations.

The command "../splrun -AN" can be used to dump the assembler code generated by the SPL compiler. Just in case you are interested.

## 3.32 SPL Performance

Variable values in SPL are always stored in so-called SPL Nodes ('struct spl_node' in spl.h). Such an SPL node is a heavy data structure, aprox. 100 bytes large. It has fields for a wide range of value types and some additional fields for internal purposes (example given garbage collection).

The size, complexity and propabilities of the spl_node struct is the reason why SPL is very good for handling big and complex data very fast on the one hand but small data with small low-level-operations very slow on the other hand.

Example given: Most scripts do a lot of string concatenations. SPL is internally storing all strings as binary trees so all the malloc-copy-free cycles found in other scripting languages are not required and string concatenation is a pretty fast operation in SPL. But small operations such as incrementing an integer are extreamly slow in SPL. Two extreme examples:

Damn slow in SPL compared to other scripting languages:

```
for (var i=0; i < 100000; i++) { /* do nothing */ }
```

Damn fast in SPL compared to other scripting languages:

```
var text = "foobar";
for (var i=0; i < 1000; i++) text = "$text$text";
```

The 1st example does nothing else than incrementing a counter up to 100000. This is pretty slow in SPL because the increment operator creates a new value (i.e. a new spl_node struct) and frees the old value for every increment operation. Other scripting languages would simply overwrite the integer pointed to by the variable 'i'. This is not possible in SPL because 'i' points to an spl_node and this node is strictly read only because other variables may also point to it and so changeing it would also change this other variables.

The 2nd example would try to allocate about 6^1000 bytes of memory (about 1.416e+769 Gigabytes) in other languages. But SPL never tries to actually allocate the memory because in this example program the string is only used in other string concatenations.

Both examples are extreme and not representative for real world applications. In general one should try to do the "big logic" in SPL but implement the inner loops of performance critical algorithms somewhere else (example given in SPL functions written in C). The last chapter of the SPL manual (i.e. the file README.API) handles how to use the SPL C-API, example given for writing SPL modules in C.

## 3.33 Command Line Debugger

The command line debugger is still under construction.

# Chapter 4

# WebSPL and WSF Tutorial

This is a short tutorial to WebSPL and WSF. It is not a reference manual, so also have a look at the "wsf" module, the "cgi" module and related modules in the module references.

If you have not yet set up your Web Server to handle webspl scripts, please have a look at the README file or the 'What is SPL?' chapter in this manual.

## 4.1 WebSPL Introduction

WebSPL is an SPL runtime for CGI programming. There are two implementations of WebSPL: The CGI script "webspl.cgi", for integrating WebSPL with the Apache webserver using an 'action-handler', and "webspld", a stand-alone HTTP server.

The SPL sources inlcude an example "httpd.conf" file for using "webspl.cgi" with the Apache webserver. The "webspld" program prints its own usage info when called with '−−help'.

The interesting thing about WebSPL is, that, other than usual CGI scripts, a WebSPL script is not executed once for each HTTP request. Instead, there is just one (virtual) SPL process for a session and the WebSPL script can pause it's execution at any time, wait for the user to do something and continue execution as soon as the webbrowser sends the next HTTP request for the session.

This is a tutorial - so here is our first example program:

```
/* -- webspltut/webspltut00.webspl -- */

load "task";
load "cgi";

for (var i=0; i<10; i++) {
      write(<>
            [$i] Hello World!  
            <b>[<a href="$cgi.url?sid=$cgi.sid">next</a>]</b>
      </>);
      task_pause();
}
```

```
write("That's it!\n");
```

Whenever task_pause() is called, the program execution stops and what has been written since the last task_pause() (or the beginning of the program execution) is displayed in the web browser.

As soon as the user reacts (i.e. clicks on "next" in this example), the execution of the program is continued and task_pause() returns. The query string parameter 'sid' is special: It always contains the session id and must not be used for anything else in WebSPL applications. If the script is using SPL tasks, it is also possible to include the ID of the task which should be woken up in the 'sid' parameter. E.g. WSF is doing that.

## 4.2 WSF Introduction

WSF (WebSPL Forms) is a library for doing web application development with WebSPL. WSF provides a widget based interface and so allows the development of web applications with a look&feel like normal GUI programs. But it is a bad choice for creating dynamic webpages since example given the browser back button must not be used in WSF applications. For this (and other) reasons it is recommended to open WSF applications in browser popup windows without browser toolbars.

There are two base classes in WSF: WsfComponent and WsfDocument.

The WsfDocument object handles a browser window. Usually there is only one instance of that object in a program, but if you want to use WSF to manage multiple windows, one WsfDocument instance per browser window is required. This WsfDocument instance is usually (i.e. per convention, there is no technical reason for doing so) assigned to a global variable named "page".

The content of a browser window is organised as a tree of WsfComponent objects. The root component must be assigned to the "root" member variable of the "page" object. The WsfComponent object itself has a member variable called "children" which is an array/hash of the child components.

In most cases objects derived from WsfComponent are used instead of WsfComponent directly. One of these derived objects is WsfDisplay from the "wsf_display" module (WsfDocument and WsfComponent are declared in the "wsf" module).

The HTML text which should be displayed in a WsfDisplay object is simply passed to the constructor. So here is a very simple WSF "Hello World" application:

```
/* -- webspltut/webspltut01.webspl -- */

load "wsf";
load "wsf_display";

var page = new WsfDocument();
page.root = new WsfDisplay("Hello World!");
page.main();
```

The page.main() does never return.

The default behavior of the WsfComponent object is to simply print the html content of all its children in its get_html() method. So it is pretty useful as a simple container for other more complex objects derived from WsfComponent:

```
/* -- webspltut/webspltut02.webspl -- */

load "wsf";
load "wsf_display";

var page = new WsfDocument();
page.root = new WsfComponent();
page.root.children[0] = new WsfDisplay("Hello");
page.root.children[1] = new WsfDisplay("World!");
page.main();
```

But this application doesn't do much. So, lets create our own object derived from WsfComponent which implements a simple counter:

```
/* -- webspltut/webspltut03.webspl -- */

load "wsf";

object Counter WsfComponent
{
      var counter_value = 100;

      method get_html()
      {
            return
            <>
                  <div id="$id">
                  <b>Hello World! [$counter_value]</b>
                  <br/>
                  <a ${add_href("${cgi.url}?sid=${sid}&mode=up")}>Up</a>
                  <a ${add_href("${cgi.url}?sid=${sid}&mode=down")}>Down</a>
                  </div>
            </>;
      }

      method main()
      {
            while (1) {
                  task_co_return();

                  if (declared cgi.param.mode &&
                      cgi.param.mode ~== "up")
                        counter_value++;

                  if (declared cgi.param.mode &&
                      cgi.param.mode ~== "down")
                        counter_value--;
```

```
                dirty = 1;
            }
        }
}

var page = new WsfDocument();
page.root = new Counter();
page.main();
```

So, when creating your own WSF Components, you need to overload the get_html() method with something which returns the HTML code for your component and overload the main() method with the main program for the task managing this component. The task executing the main() method is created automatically by the object constructor (and killed by the destroy() method). The function task_co_return() must be called by main() to wait for user events. The variable 'dirty' must be set to 1 when anything happend with an effect on the get_html() output. The root element of the HTML code generated by get_html() must have its 'id' attribute set to the value of the 'id' variable. There are methods (such as the add_href() used in the example) for creating links, formulas, etc.

Have a look at the module reference for the "wsf" module for a full overview of the WsfDocument and the WsfComponent objects.

### 4.2.1 Adding Menus

Applications always have some type of menu. There is a generic WSF component called WsfMenu (from the "wsf_menu" module) which implements nice menus. The menu entries are function pointers which are called when the entry has been clicked:

```
/* -- webspltut/webspltut04.webspl -- */

load "wsf";
load "wsf_display";
load "wsf_menu";

var menu = [
      "Foo" => [
            "Entry 1" => function() {
                  page.root.child_set(1, new WsfDisplay("Foo Entry 1"));
            },
            "Entry 2" => function() {
                  page.root.child_set(1, new WsfDisplay("Foo Entry 2"));
            },
            "Entry 3" => function() {
                  page.root.child_set(1, new WsfDisplay("Foo Entry 3"));
            }
      ],
      "Bar" => function() {
```

```
            page.root.child_set(1, new WsfDisplay("This is Bar"));
      }
];

var page = new WsfDocument();
page.root = new WsfComponent();
page.root.children[0] = new WsfMenu(menu);
page.main();
```

The HTML code generated by WsfMenu should be the first real code in the webpage. So the example is using a simple WsfComponent as root component and WsfMenu as its first child (0). The 2nd child (1) is set to WsfDisplay objects by the menu callbacks to display messages.

### 4.2.2 Example Application: A very simple CMS

Now lets write a real application: a very simple CMS. Don't expect too much: It is not much more than just a web-based file manager for HTML files. Do not wonder: the login for the login screen is covered in the next section.

```
/* -- webspltut/webspltut05.webspl -- */

load "wsf";
load "wsf_display";
load "wsf_menu";

load "file";
load "encode_js";

object Editpage WsfComponent
{
      var filename;

      method get_html()
      {
            var data = file_read(filename);

            return
            <>
            <div style="background-color: #aaaaaa; width:86%; height:80%;
                  position:absolute; left:5%; top:10%; padding:2%"
                  id="$id">
            <h3>Edit page '${xml::filename =~ s/\.html$//R}':</h3>
            <form ${add_action(cgi.url)}>
            <input type="hidden" name="sid" value="${xml::sid}">
            <textarea style="width:100%; height:80%;"
                  name="data">${xml::data}</textarea>
            <br/>
            <input type="submit" value="Save">
            <input type="button" value="Show"
```

```
                onClick="window.open('${js::filename}', '');">
            <input type="submit" name="remove" onClick="return confirm(
                '${js::"Do you really want to remove page $[
                ]`${filename =~ s/\.html$//R}`?"}');"
                value="Remove Page">
            </form>
            </div>
            </>;
        }

        method main()
        {
            task_co_return();
            if (declared cgi.param.remove) {
                file_delete(filename);
                create_menu();
                page.root.child_set("content", new WsfComponent());
            } else {
                file_write(filename, cgi.param.data);
                dirty = 1;
            }
        }

        method init(_filename)
        {
            filename = _filename;
            return *WsfComponent.init();
        }
}

object Newpage WsfComponent
{
        method get_html()
        {
            return
            <>
            <div style="background-color: #aaaaaa; width:30%;
                position:absolute; left:30%; top:20%; padding:10px"
                id="$id">
            <h3>Create a new page:</h3>
            <form ${add_action(cgi.url)}>
            <input type="hidden" name="sid" value="${xml::sid}">
            <b>Filename:</b><br />
            <input style="width:100%;" name="filename" /><p />
            <b>Page Title:</b><br />
            <input style="width:100%;" name="pagetitle" /><p />
            <input type="submit" value="Create">
            </form>
            </div>
            </>;
```

```
	}

	method main()
	{
		task_co_return();

		var pagetitle = xml::cgi.param.pagetitle;
		var filename  = xml::cgi.param.filename;

		filename =~ s/^(.*\/|)\.*//;
		filename =~ s/(?<!\.html)$/.html/;

		var html = <<EOT:
			:<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
			:<html>
			:<head>
			:  <meta http-equiv="content-type" content="text/html; charset=UTF
			:  <title>${xml::pagetitle}</title>
			:<head>
			:<body>
			:  <h1>${xml::pagetitle}</h1>
			:</body>
			:</html>
		EOT;

		file_write(filename, html);

		create_menu();
		page.root.child_set("content", new Editpage(filename));
	}
}

function create_menu()
{
	var menu;

	menu["Pages"] = ({
		var pages;
		var ls = file_list(".");
		foreach i (ls)
			if ( ls[i] =~ /^(?P<filename>(?P<title>.*)\.html)$/ ) {
				import $$;
				pages[title] = function() {
					page.root.child_set("content",
						new Editpage(filename));
				};
			}
		return pages;
	});
```

```
        menu["New Page"] = function() {
              page.root.child_set("content", new Newpage());
        };

        page.root.child_set("menu", new WsfMenu(menu));
}

// hide login logic because it is covered in a different tutorial section
#file-as-code websplutut06.spl

var page = new WsfDocument();
page.root = new WsfComponent();
create_menu();
page.main();
```

Please run the application before you continue reading this tutorial (just to make sure you know what this is all about).

The application defines two Objects: Editpage for dialog which lets the user edit a page and Newpage for the dialog which lets the user create new pages.

The application also defines the function create_menu() for (re-)creating the menu.

The main program just creates the page object, the root component, calls create_menu() and enters page.main().

### The create_menu() function

This function creates the menu as child component of the root component. The "Pages" menu is dynamically generated from the list of *.html files in the current directory. Whenever files are added or removed, create_menu() must be called in order to update this menu.

A closure is used to store the filename to be opened together with the callback function which is called when an entry in the "Pages" menu is clicked.

### The Editpage Object

Editpage is directly dervied from WsfComponent. The filename of the file to be edited is passed as an argument to the constructor and copied to the variable 'filename' there. The methods get_html() and main() are overloaded.

The get_html() method simply displays an HTML form for editing the page. The page content is loaded from the HTML file. Note how the 'xml::' and 'js::' encoding functions are used to ensure the correctness of the generated HTML code.

The main() method simply writes back the data to the file, or removes the file if the user clicked on the "Remove" button. Since main() is automatically called in an endless-loop, I did not add the "while (1) { ... }" loop here explicitly. When main() is removing the file, the menu is updated by calling create_menu() and the Editpage component is replacing itself with a simple WsfComponent instance.

**The Newpage Object**

The Newpage opject (also directly dervied from WsfComponent) is even simpler. get_html() just displays an HTML form again. When the form is submitted and main() returns from task_co_return(), it simply creates the new page (after cleaning up the filename), calls create_menu() and replaces itself with an Editpage instance for the newly created file.

Note that a Here-Document with indenting character is used when creating the HTML text for the new page. This avoids ugly leading blanks in the HTML file while keeping the indenting in the program intact.

### 4.2.3 Login screen and opening a popup window

The last example is using a pretty nice mechanism for doing authentication and opening a popup window. Before creating the 'page' object and passing control to WSF, this code is executed:

```
/* -- webspltut/webspltut06.spl -- */

// this is included from webspltut05.webspl before creating the 'page' object.

if (not declared cgi.param.user or cgi.param.user ~!= "demo" or
    not declared cgi.param.pass or cgi.param.pass ~!= "demo") {
          write(<>
            <p/> 
            <H1 align="center">Login</H1>
            <div align="center">
                  <b>Username and password are both "demo".</b>
            </div>
            <p/>
            <form method="post">
            <table align="center">
            <tr><td>Username:</td>
                <td><input style="width: 100%" size="20" name="user"
                  value="demo" /></td></tr>
            <tr><td>Password:</td>
                <td><input style="width: 100%" size="20" name="pass"
                  type="password" value="demo" /></td></tr>
            <tr><td colspan="2">
                  <input type="checkbox" name="nopopup" />
                  <span onClick="nopopup.checked = !nopopup.checked">
                        Do not open a popup window.
                  </span></td></tr>
            <tr><td colspan="2" align="right">
                  <input type="submit" value="Login" /></td></tr>
            </table>
            </form>
      </>);
      exit;
}
```

```
if (not declared cgi.param.nopopup) {
      write(<>
      <script><!--
      window.open("${cgi.url}?sid=${cgi.sid}&start=${
            declared cgi.param.start ? cgi.param.start : ''}", "",
            "hotkeys=no,location=no,menubar=no,height=600,width=800," +
            "resizable=yes,scrollbars=yes,status=no,toolbar=no");
      //--></script>
      </>);

      task_pause();
}
```

There is nothing wrong with already accepting parameters at program startup. The program expects username and password in the query string parameters "user" and "pass". If they are not present or wrong, the login screen will be printed and the program exits.

Only if correct authentication tokens are passed, the program continues execution and initializes the WSF framework. It has shown that this approach is very useful and much easier than actually including this step in the application itself. One of the benefits of this approach is that a developer can pass the authentication tokens already in the query string and so bypass the additional screen. That's very comfortable for testing the application during the development. Some applications even allow 'deep-linking' to some dialogs this way to make testing easier. This approach also allows to open the application in a popup window easily.

Opening a popup window makes sense since we do not want the user to click on the browser 'back' button or bookmark any deep links of the application. It doesn't really prevent the user from doing such stuff - but it makes it hard enough to avoid most of the troubles.

Opening the popup window is simply done by printing the javascript code and then calling task_pause(). As soon as the the browser tries to load the content of the popup window, the session is resumed and task_pause() returns.

## 4.3 W2T Introduction

The W2T (Web 2.0 Toolkit) module provides another framework for web application development. W2T applications only send an initial XML document (usually XHTML or SVG) to the browser and everything after is done using AJAX XML RPC calls.

The W2T module is basically a collection of SPL and JavaScript functions which provide functionality for updating the XML DOM tree displayed in the browser window from server-side SPL and call server-side SPL event callbacks from browser-side JavaScript code.

Here is a small W2T example program which is using SVG. Note that at the time of this writing Firefox 1.5 is the only browser which can be used as frontend for W2T SVG programs:

```
/* -- webspl_demo/w2tdemo_svg.webspl -- */
```

```
load "w2t";

var page = new W2t();

var xpos = 50;

page.register_callback("clicked_on_box",
function(xmlnode, word1, word2)
{
      debug "Pushed the button: $word1 $word2";

      var newboxid = page.getid();

      function newboxclick() {
            page.response_add(<:>
                  : <dom method="remove" on="$newboxid" />
            </>);
      }

      page.register_callback('${newboxid}_click', newboxclick);
      page.response_add(<:>
            : <dom method="appendchild" on="root">
            :      <rect xmlns="http://www.w3.org/2000/svg"
            :            id="$newboxid" x="$xpos" y="30"
            :            width="10" height="10" fill="blue"
            :            w2t:callbacks="${newboxid}_click"
            :            onclick="w2t_callback('${newboxid}_click')" />
            : </dom>
      </>);

      xpos += 20;
});

// Set startup page
page.startup = <:>
      : <?xml version="1.0" ?>
      : <svg xmlns="http://www.w3.org/2000/svg" xmlns:w2t="$page.w2t_ns" id="root">
      :
      : <script><![CDATA[
      :
      : ${page.getscripts()}
      :
      : ]]></script>
      :
      : <rect x="30" y="30" width="10" height="10" fill="red"
      :      onclick="w2t_console_open()" />
      :
      : <rect x="100" y="100" width="100" height="100" rx="10" ry="10"
      :       fill="yellow" stroke="navy" stroke-width="10"
      :      w2t:callbacks="clicked_on_box"
```

```
        :      onclick="w2t_callback('clicked_on_box', ['hello', 'world'])" />
        :
        : </svg>
    </>;

    page.run();
```

The W2T module also includes some JavaScript helper functions for doing animations and other funny stuff. W2T is under development and a more detailed documentation will be provided when it is finished.

# Chapter 5

# Builtins Function Library

This module contains the SPL 'standard' builtin functions. It does not need to be loaded explicitely, most SPL runtimes load it automatically for you.

## 5.1 builtin write(text, encoding);

Write the text passed as argument to standard ouput. Some SPL runtimes replace this function with their own output function.

If the output should not be UTF-8 encoded, the encoding must be specified with a 2nd parameter. Valid encodings are the same as for the '#encoding' compiler pragma (see 'SPL Language Reference' for a list).

## 5.2 builtin read(prompt, encoding);

Read a line from the standard input. Some SPL runtimes replace this function with their own input function. The read text (or undef on end-of-file) is returned.

The fist parameter can be used to specify a prompt to be displayed for reading the line.

If the output is not UTF-8 encoded, the encoding must be specified with a 2nd parameter. Valid encodings are the same as for the '#encoding' compiler pragma (see 'SPL Language Reference' for a list).

If UTF-8 encoding is expected but the output fails to pass the UTF-8 test, the output is assumed to be latin1 encoded.

## 5.3 builtin chr(number);

Create a (unicode) charater from the number passed as argument.

## 5.4 builtin ord(text);

Return the unicode character number of the first character in the text.

## 5.5 builtin hex(text);

Convert a string holding a hexadezimal number to an integer value. A leading "0x" or "0X" is ignored. Zero is returned if there is no hexadezimal number in the string.

## 5.6 builtin oct(text);

Convert a string holding an octal number to an integer value. A leading "0o" or "0O" is ignored. Zero is returned if there is no octal number in the string.

## 5.7 builtin bin(text);

Convert a string holding a binary number to an integer value. A leading "0b" or "0B" is ignored. Zero is returned if there is no binary number in the string.

## 5.8 builtin fmt(text, @args);

A C printf()-like format function. The formatted string is returned. The following special sequences are supported:

```
%%      A '%' character
%s      A string
%d      A decimal integer
%x      A hexadecimal integer (lowercase letters)
%X      A hexadecimal integer (uppercase letters)
%o      An octal integer
%b      A binary integer
%f      A floating point number
```

The field width and precision format of printf() is also supported. But currently there is no support for 'N$' argument specifications.

## 5.9 builtin rand(max);

Returns a random integer number, larger then or equal to zero and smaller then the 'max' argument.

When the argument is 0 or missing, the return value is a floating point number between 0 and 1. When the argument is -1, the return value is an integer between 0 and RAND_MAX.

## 5.10 builtin setlocale(category, locale);

The setlocale() function is used to set or query the program's current locale.

This is a simple wrapper for the C setlocale() function. The 'category' argument is passed as string. E.g.:

```
setlocale("LC_ALL", "C");
```

## 5.11 builtin bindtextdomain(domain, directory);

The bindtextdomain() function sets the base directory of the hierarchy containing message catalogs for a given message domain.

This is a simple wrapper for the C bindtextdomain() function.

## 5.12 builtin textdomain(domain);

The textdomain() function sets or retrieves the current message domain.

This is a simple wrapper for the C textdomain() function.

## 5.13 builtin gettext(message);

The gettext() function attempts to translate a text string into the user's native language, by looking up the translation in a message catalog.

This is a simple wrapper for the C gettext() function.

## 5.14 builtin dgettext(domain, message);

The dgettext() function attempts to translate a text string into the user's native language, by looking up the translation in a message catalog.

This is a simple wrapper for the C gettext() function.

## 5.15 builtin _(message, domain, @args);

This function is implementing the backend functionality of the SPL translation prefix for strings (_).

It translates the message using the gettext() function and then substitutes '{N}' with the strings passed as additional arguments (N beeing the index in @args) and '{}' with a simple '{'.

## 5.16 builtin acos(x);

arc cosine function

## 5.17 builtin asin);

arc sine function

## 5.18 builtin atan(x);

arc tangent function

## 5.19 builtin atan2(x, y);

arc tangent function of two variables

## 5.20 builtin cos(x);

cosine function

## 5.21 builtin sin(x);

sine function

## 5.22 builtin tan(x);

tangent function

## 5.23 builtin sqrt(x);

square root function

## 5.24 builtin round(x);

round to nearest integer

## 5.25 builtin ceil(x);

smallest integral value not less than argument

## 5.26 builtin floor(x);

largest integral value not greater than argument

# Chapter 6

# Module "array"

Array Module

This module extends the built-in functionality of SPL for working with arrays. Please also have a look at the Language Reference Manual for a description of the built-in operators provided natively by SPL.

## 6.1 builtin array_reindex(array);

This function re-indexes an array. That means, the order of the elements is kept as they are, but the keys are newly assigned (using integer, starting with 0).

## 6.2 builtin array_switch(array, key1, key2);

This function switches two elements of an array. The keys and values are not modified by this - only the order of the elements in the array.

## 6.3 function array_sort_by_keys(array, order_function);

Sorting by keys

This function sorts the passed array ussing the passed order function. The order function is called whenever this function wants to compare two elements. The key of the 1st element is passed as 1st parameter to the order_function. The key of the 2nd element as 2nd parameter. The order function must return 1 if the elements are in the wrong order.

Example:

```
var a = [ 5 => 'a', '3' => 'b', '8' => 'c' ];

array_sort_by_keys(a, function(a,b) { return a > b; });
```

```
foreach i (a)
      debug "$i --> ${a[i]}";
```

This creates the output:

```
SPL Debug: 3 --> b
SPL Debug: 5 --> a
SPL Debug: 8 --> c
```

Small arrays (less then 10 elements) are sorted using a simple bubble sort. Big arrays are sorted using the system qsort() function (if available) and using a btree sort otherwise.

## 6.4 function array_sort_by_values(array, order_function);

Sorting by values

This function is pretty simmilar to array_sort_by_keys. The only difference is that it passes references to the elements which should be compared to the order function.

## 6.5 function array_join(array, seperator);

Concatenate all array elements to a string

# Chapter 7

# Module "bits"

Bits Module
: This module adds built-in functions for bit manipulations.

## 7.1 builtin bits_and(@values);

Do a bitwise AND between all arguments and return the result.

## 7.2 builtin bits_or(@values);

Do a bitwise OR between all arguments and return the result.

## 7.3 builtin bits_xor(@values);

Do a bitwise XOR between all arguments and return the result.

## 7.4 builtin bits_not(value);

Do a bitwise NOT of the argument and return the result.

## 7.5 builtin bits_shl(value, bits);

Do a bitwise SHIFT-LEFT of the argument and return the result.

## 7.6 builtin bits_shr(value, bits);

Do a bitwise SHIFT-RIGHT of the argument and return the result.

# Chapter 8

# Module "cgi"

Module for writing CGIs

## 8.1 builtin cgi_userfile_save(paramname, filename);

This function can be used to save an uploaded file to the local filesystem. The 1st parameter is the name of the upload form field and the 2nd parameter the local filename.

The file will be truncated if it exists already and created if it does not.

The pseudo-variable 'cgi.param.<name>' contains the name of the file. The content of the uploaded file can't be accessed directly.

This function returns the number of bytes written to the harddisk or undef for an error.

## 8.2 builtin cgi_write(text);

This function sends the specified text to the web browser. It also creates the HTTP header when it is called the first time for an HTTP request.

## 8.3 namespace cgi

This namespace holds all the CGI specific data, like query string parameters or browser type.

### 8.3.1 var cgi.param;

A hash with all query string parameters, parameter name as key and parameter value as value.

This variable is read-only.

### 8.3.2 var cgi.cookie;

A hash with all cookies passed by the browser. Writing to this hash is only possible as long as no cgi_write() has been isued.

### 8.3.3 var cgi.config;

A hash with all config variables read from the webspl.conf files. Example given:

```
# In webspl.conf
[ myapp.webspl ]
myapp.dbtype = sqlite
myapp.database = /var/lib/myapp/database.bin

// In the script
var db = sql_connect(cgi.config.myapp.dbtype, cgi.config.myapp.database);
```

### 8.3.4 var cgi.content_type = "text/html";

The HTTP content-type of the document to be send back to the browser. This variable is write-only and must be set before cgi_write() is called the first time. It is automatically reset to "text/html" for each new HTTP response.

### 8.3.5 var cgi.silent_debug = 0;

When this variable has a non-zero value, the output created by debug statements is not sent to the browser. This variable is write-only. It is automatically reset to '0' for each new HTTP response.

### 8.3.6 var cgi.sid;

The current session id (with the now running task)

This variable is read-only.

### 8.3.7 var cgi.sid_vm;

The current session id (without task)
This variable is read-only.

### 8.3.8 var cgi.sid_task;

The task part from the session id requested by the user.
This variable is read-only.

### 8.3.9 var cgi.sid_passed;

The session id as requested by the user (vm and task).
This variable is read-only.

### 8.3.10 var cgi.url;

The url requested by the browser (without the query string). This is useful for building self-references.
This variable is read-only.

### 8.3.11 var cgi.agent;

The identification string sent by the user agent.
This variable is read-only.

### 8.3.12 var cgi.peerip;

The IP address of the peer.
This variable is read-only.

### 8.3.13 var cgi.post_type;

The mime type of the data sent in a POST request.
This variable is read-only.

### 8.3.14 var cgi.post_data;

The data sent in a POST request when it is a text/<something> mime type.
This variable is read-only.

# Chapter 9

# Module "crypt"

A crypt() function (like in C).

## 9.1 builtin crypt(key, salt);

This function is a frontend to the C crypt() function. If salt is not given or shorten then 2 characters then a random salt ([a-zA-Z0-9./]{2}) is used.

# Chapter 10

# Module "curl"

cURL Module

This module adds an interface to the cURL library.
WARNING: This module is still under construction.

## 10.1 builtin curl(url, %options);

Perform a cURL file transfer

The first argument to this function is the URL. All other arguments (cURL options) must be specified by name (lowercase and without the CURLOPT_ prefix). A full list of cURL options can be found at:

```
http://curl.haxx.se/libcurl/c/curl_easy_setopt.html
```

The return value is a data structure with the following fields:

```
.header
      The HTTP response header

.body
      The actual HTTP response
```

Example given:

```
var result = curl("http://de.wikipedia.org/wiki/Spezial:Search?",
            post: 1, postfields: "search=SPL_Programming_Language");
debug result.header;
```

# Chapter 11

# Module "encode_int"

Integer encoder

## 11.1 function encode_int(text);

This function gets a text as parameter and returns the portion of the text from the beginning which only consists of numbers. Deciamal numbers and negative numbers are not supported. If there is no number at the beginning of the passed text, 0 is returned.

Whitespaces before the integer are ignored.

The primary use of this function is to 'quote' integers in those cases where it is not possible to do real quoting but a user input needs to be quoted somehow for security. E.g. SQLite handles an integer differently when it is quoted as text and so quoting using encode_sql() would work out.

This function is designed to be used with the encoding/quoting operator (::).

# Chapter 12

# Module "encode_js"

Javascript encoder

## 12.1 function encode_js(text);

This function encodes a string so it is perfectly quoted for javascript. The characters \, ', ", &, <, >, newline and tab are encoded using \ooo sequences. The resulting string is _not_ quoted because it is hard to tell for this function if you prefer single or double quotes.

This function is designed to be used with the encoding/quoting operator (::).

# Chapter 13

# Module "encode_regex"

Regex encoder

## 13.1 function encode_regex(text);

This function gets a text as parameter and returns the same text with all characters escaped which do have a special meaning in regular expressions.

The primary use of this function is to include a text in a regular expression as it is.

This function is designed to be used with the encoding/quoting operator (::).

# Chapter 14

# Module "encode_url"

URL encoder

## 14.1 function encode_url(text);

This function URL encodes a string. All charachters except [a-zA-Z0-9] are encoded.

This function is designed to be used with the encoding/quoting operator (::).

# Chapter 15

# Module "encode_xml"

A simple XML encoding function

## 15.1 builtin encode_xml(text);

This function ecodes a text in XML. I.e. the characters &, <, >, " and ' are translated to &, <, >, ", and '.

This function is designed to be used with the encoding/quoting operator (::).

# Chapter 16

# Module "environ"

A module for accessing environment variables

## 16.1 namespace environ

This namespace contains all environment variables. It is possible to create, modify and remove entries:

```
environ.PATH ~= ":${environ.HOME}/bin";
```

# Chapter 17

# Module "fann"

Bindings for the FANN (Fast Artificial Neural Network) Library

## 17.1 manual introduction;

This module is a very thin wrapper for the FANN library. Thus it is possible to write applications which do leak memory or trigger segmentation faults by using this API incorrectly.

Be careful, you have been warned.

The FANN library documentation can be found on the FANN homepage:

```
http://leenissen.dk/fann/
```

The SPL FANN API is a 1:1 representation of the C API. So the original FANN documentation should answer all the questions which are left open in this document.

## 17.2 manual example;

A little FANN example app which creates a small neural network with two input and one output and trains it to act like a XOR gate.

```
load "fann";

var ann = fann_create_standard_array(3, [2, 10, 1]);

for (var i=0; i<1000; i++)
{
    fann_train(ann, [0, 0], [0]);
    fann_train(ann, [0, 1], [1]);
    fann_train(ann, [1, 0], [1]);
    fann_train(ann, [1, 1], [0]);
```

```
}

for (var a=0; a<2; a++)
for (var b=0; b<2; b++)
{
    var result = pop fann_run(ann, [a, b]);
    debug "$a XOR $b <=> ${round(result)}  [${fmt("%5.03f", result)}]";
}

fann_destroy(ann);
```

Note that the fann_destroy() call is not absolutely needed because fann_destroy() is automatically called when the 'ann' variable is removed by the garbage collector.

## 17.3 manual function_list;

The following FANN functions are available via this SPL module:

```
fann_cascadetrain_on_data(ann, data, max_neurons, neurons_between_reports, desired_e
fann_cascadetrain_on_file(ann, filename, max_neurons, neurons_between_reports, desir
fann_create_from_file(configuration_file);
fann_create_shortcut_array(num_layers, layers);
fann_create_sparse_array(connection_rate, num_layers, layers);
fann_create_standard_array(num_layers, layers);
fann_destroy(ann);
fann_destroy_train(train_data);
fann_duplicate_train_data(data);
fann_get_MSE(ann);
fann_get_bit_fail(ann);
fann_get_bit_fail_limit(ann);
fann_get_cascade_activation_functions(ann);
fann_get_cascade_activation_functions_count(ann);
fann_get_cascade_activation_steepnesses(ann);
fann_get_cascade_activation_steepnesses_count(ann);
fann_get_cascade_candidate_change_fraction(ann);
fann_get_cascade_candidate_limit(ann);
fann_get_cascade_candidate_stagnation_epochs(ann);
fann_get_cascade_max_cand_epochs(ann);
fann_get_cascade_max_out_epochs(ann);
fann_get_cascade_num_candidate_groups(ann);
fann_get_cascade_num_candidates(ann);
fann_get_cascade_output_change_fraction(ann);
fann_get_cascade_output_stagnation_epochs(ann);
fann_get_cascade_weight_multiplier(ann);
fann_get_errno(errdat);
fann_get_errstr(errdat);
fann_get_learning_momentum(ann);
```

```
fann_get_learning_rate(ann);
fann_get_num_input(ann);
fann_get_num_output(ann);
fann_get_quickprop_decay(ann);
fann_get_quickprop_mu(ann);
fann_get_rprop_decrease_factor(ann);
fann_get_rprop_delta_max(ann);
fann_get_rprop_delta_min(ann);
fann_get_rprop_increase_factor(ann);
fann_get_total_connections(ann);
fann_get_total_neurons(ann);
fann_get_train_error_function(ann);
fann_get_train_stop_function(ann);
fann_get_training_algorithm(ann);
fann_init_weights(ann, train_data);
fann_length_train_data(data);
fann_merge_train_data(data1, data2);
fann_num_input_train_data(data);
fann_num_output_train_data(data);
fann_print_connections(ann);
fann_print_error(errdat);
fann_print_parameters(ann);
fann_randomize_weights(ann, min_weight, max_weight);
fann_read_train_from_file(filename);
fann_reset_MSE(ann);
fann_reset_errno(errdat);
fann_reset_errstr(errdat);
fann_run(ann, input);
fann_save(ann, configuration_file);
fann_save_to_fixed(ann, configuration_file);
fann_save_train(data, filename);
fann_save_train_to_fixed(data, filename, decimal_point);
fann_scale_input_train_data(train_data, new_min, new_max);
fann_scale_output_train_data(train_data, new_min, new_max);
fann_scale_train_data(train_data, new_min, new_max);
fann_set_activation_function(ann, activation_function, layer, neuron);
fann_set_activation_function_hidden(ann, activation_function);
fann_set_activation_function_layer(ann, activation_function, layer);
fann_set_activation_function_output(ann, activation_function);
fann_set_activation_steepness(ann, steepness, layer, neuron);
fann_set_activation_steepness_hidden(ann, steepness);
fann_set_activation_steepness_layer(ann, steepness, layer);
fann_set_activation_steepness_output(ann, steepness);
fann_set_bit_fail_limit(ann, bit_fail_limit);
fann_set_cascade_activation_functions(ann, cascade_activation_functions, cascade_act
fann_set_cascade_activation_steepnesses(ann, cascade_activation_steepnesses, cascade
fann_set_cascade_candidate_change_fraction(ann, cascade_candidate_change_fraction);
fann_set_cascade_candidate_limit(ann, cascade_candidate_limit);
fann_set_cascade_candidate_stagnation_epochs(ann, cascade_candidate_stagnation_epoch
fann_set_cascade_max_cand_epochs(ann, cascade_max_cand_epochs);
```

```
fann_set_cascade_max_out_epochs(ann, cascade_max_out_epochs);
fann_set_cascade_num_candidate_groups(ann, cascade_num_candidate_groups);
fann_set_cascade_output_change_fraction(ann, cascade_output_change_fraction);
fann_set_cascade_output_stagnation_epochs(ann, cascade_output_stagnation_epochs);
fann_set_cascade_weight_multiplier(ann, cascade_weight_multiplier);
fann_set_error_log(errdat, log_file);
fann_set_learning_momentum(ann, learning_momentum);
fann_set_learning_rate(ann, learning_rate);
fann_set_quickprop_decay(ann, quickprop_decay);
fann_set_quickprop_mu(ann, quickprop_mu);
fann_set_rprop_decrease_factor(ann, rprop_decrease_factor);
fann_set_rprop_delta_max(ann, rprop_delta_max);
fann_set_rprop_delta_min(ann, rprop_delta_min);
fann_set_rprop_increase_factor(ann, rprop_increase_factor);
fann_set_train_error_function(ann, train_error_function);
fann_set_train_stop_function(ann, train_stop_function);
fann_set_training_algorithm(ann, training_algorithm);
fann_shuffle_train_data(train_data);
fann_subset_train_data(data, pos, length);
fann_test(ann, input, desired_output);
fann_test_data(ann, data);
fann_train(ann, input, desired_output);
fann_train_epoch(ann, data);
fann_train_on_data(ann, data, max_epochs, epochs_between_reports, desired_error);
fann_train_on_file(ann, filename, max_epochs, epochs_between_reports, desired_error)
```

## 17.4 builtin fann_enum(enum_name);

This function converts a FANN enum name to its integer value.

## 17.5 object FannEx

An instance of this object is thrown on errors. Note that this exception object is only used for errors in the FANN<->SPL bindings. Errors which happened in FANN are handled using the usual FANN error handling mechanism.

### 17.5.1 var FannEx.description;

A description text describing the error.

# Chapter 18

# Module "file"

This module provides a simple API for reading and writing files

## 18.1 builtin file_read(filename, encoding);

This function reads the specified filename and returns the file content.

If the file is not UTF-8 encoded, the encoding must be specified with a 2nd parameter. Valid encodings are the same as for the '#encoding' compiler pragma (see 'SPL Language Reference' for a list).

A FileEx exception is thrown on I/O errors.

## 18.2 builtin file_write(filename, content, encoding);

This function writes the specified content to the specified filename. The file is created if neccassary and overwritten if it is already there.

If the file should not be UTF-8 encoded, the encoding must be specified with a 3rd parameter. Valid encodings are the same as for the '#encoding' compiler pragma (see 'SPL Language Reference' for a list).

A FileEx exception is thrown on I/O errors.

## 18.3 builtin file_delete(filename);

This function removes the specified file. A FileEx exception is thrown on I/O errors.

## 18.4 builtin file_list(dirname);

This function returns the list of files in the specified directory. A FileEx exception is thrown on I/O errors.

## 18.5 builtin file_access(filename, mode);

This function returns true if the file can be accessed in the specified mode. The mode argument is a string that may contain the characters 'R', 'W', 'X' and 'F'.

## 18.6 object FileEx

An instance of this object is thrown on file I/O errors.

### 18.6.1 var FileEx.type;

Specifies the type of operation which failed:

```
"write", "read" or "delete"
```

### 18.6.2 var FileEx.filename;

The filename of which the operation should have been performed.

### 18.6.3 var FileEx.errmsg;

The error message returned by the operating system

### 18.6.4 var FileEx.description;

A full error description including the information from type, filename and errmsg.

# Chapter 19

# Module "format_ini"

A simple INI file parser/dumper module

This module implements simple INI file parser and dumper functions.

## 19.1 function format_ini_parse(inidata);

This function expects the content of a INI file (inidata) and returns a two dimentional hash structure with the INI section names as 1st index and the INI variable name as 2nd index (initree).

The INI file format as understood by this implementation:

```
[foobar]
      the following entries are in the "foobar" section.

key = value
      set variable "key" to "value"
```

- everything else (including lines starting with ';' or '#' are ignored). - section names and variable names are limited to the regex /[a-z0-9_]+/. - variables declared before the first section declaration are

```
defined in the "" section.
```

- whitespaces are ignored.

There is no mechanism for reporting parser errors (parser errors are not possible because all lines not recognised as section or variable declaration are ignored).

## 19.2 function format_ini_dump(initree);

This function expects a data structure such as returned by format_ini_parse() as parameter and returns the inidata.

# Chapter 20

# Module "format_xml"

A simple XML parser/dumper module

This module implements simple XML parser and dumper functions.

## 20.1 builtin format_xml_parse(xmldata);

This function returns a tree of ordered hashes. The keys in the ordered hashes are encoded as following:

```
A:<Name>
      An attribute to this node.

C<n>
      Character data. <n> is counting up from zero.

E<n>:<Name>
      A child node (element) in the XML tree. <n> is counting up
      from zero. So e.g. "E0:realname" is the 1st child element
      of the type "realname".

      This is an order hash containing attributes, child nodes
      and character data again.
```

Because the hash is ordered, it is possible to get the elements in the correct order by using 'foreach' loops or using the 'next' and 'prev' instructions, or directly address the elements. E.g.

```
var xmldata =
<><?xml version="1.0" ?>
      <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
        "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
      <html>
        <head>
          <meta name="Author" content="Clifford Wolf" />
```

```
          <title>This is a simple test HTML page.</title>
          <link rel="Index" href="index.html" />
        </head>
        <body>
          <h1>Nothing interesting here!</h1>
        </body>
      </html>
</>;

var xmltree = format_xml_parse(xmldata);

debug xmltree.["E0:html"].["E0:head"].["E0:title"].["C0"];
```

## 20.2 builtin format_xml_dump(xmltree);

Create an XML text from a data structure such as returned by format_xml_parse().

## 20.3 object FormatXmlEx

An instance of this object is thrown on XML parser errors.

### 20.3.1 var FormatXmlEx.description;

A description text describing the error.

# Chapter 21

# Module "gl"

OpenGL Module
This module provides an interface to the OpenGL API.

## 21.1 manual function_list;

The following OpenGL functions are wrapped by this module:

```
void glAccum(GLenum, GLfloat);
void glActiveTexture(GLenum);
void glActiveTextureARB(GLenum);
void glAlphaFunc(GLenum, GLclampf);
void glArrayElement(GLint);
void glBegin(GLenum);
void glBindTexture(GLenum, GLuint);
void glBlendColor(GLclampf, GLclampf, GLclampf, GLclampf);
void glBlendEquation(GLenum);
void glBlendEquationSeparateATI(GLenum, GLenum);
void glBlendFunc(GLenum, GLenum);
void glCallList(GLuint);
void glClear(GLbitfield);
void glClearAccum(GLfloat, GLfloat, GLfloat, GLfloat);
void glClearColor(GLclampf, GLclampf, GLclampf, GLclampf);
void glClearDepth(GLclampd);
void glClearIndex(GLfloat);
void glClearStencil(GLint);
void glClientActiveTexture(GLenum);
void glClientActiveTextureARB(GLenum);
void glColor3b(GLbyte, GLbyte, GLbyte);
void glColor3d(GLdouble, GLdouble, GLdouble);
void glColor3f(GLfloat, GLfloat, GLfloat);
void glColor3i(GLint, GLint, GLint);
void glColor3s(GLshort, GLshort, GLshort);
void glColor3ub(GLubyte, GLubyte, GLubyte);
```

```
void glColor3ui(GLuint, GLuint, GLuint);
void glColor3us(GLushort, GLushort, GLushort);
void glColor4b(GLbyte, GLbyte, GLbyte, GLbyte);
void glColor4d(GLdouble, GLdouble, GLdouble, GLdouble);
void glColor4f(GLfloat, GLfloat, GLfloat, GLfloat);
void glColor4i(GLint, GLint, GLint, GLint);
void glColor4s(GLshort, GLshort, GLshort, GLshort);
void glColor4ub(GLubyte, GLubyte, GLubyte, GLubyte);
void glColor4ui(GLuint, GLuint, GLuint, GLuint);
void glColor4us(GLushort, GLushort, GLushort, GLushort);
void glColorMask(GLboolean, GLboolean, GLboolean, GLboolean);
void glColorMaterial(GLenum, GLenum);
void glConvolutionParameterf(GLenum, GLenum, GLfloat);
void glConvolutionParameteri(GLenum, GLenum, GLint);
void glCopyColorSubTable(GLenum, GLsizei, GLint, GLint, GLsizei);
void glCopyColorTable(GLenum, GLenum, GLint, GLint, GLsizei);
void glCopyConvolutionFilter1D(GLenum, GLenum, GLint, GLint, GLsizei);
void glCopyConvolutionFilter2D(GLenum, GLenum, GLint, GLint, GLsizei, GLsizei);
void glCopyPixels(GLint, GLint, GLsizei, GLsizei, GLenum);
void glCopyTexImage1D(GLenum, GLint, GLenum, GLint, GLint, GLsizei, GLint);
void glCopyTexImage2D(GLenum, GLint, GLenum, GLint, GLint, GLsizei, GLsizei, GLint);
void glCopyTexSubImage1D(GLenum, GLint, GLint, GLint, GLint, GLsizei);
void glCopyTexSubImage2D(GLenum, GLint, GLint, GLint, GLint, GLint, GLsizei, GLsizei
void glCopyTexSubImage3D(GLenum, GLint, GLint, GLint, GLint, GLint, GLint, GLsizei,
void glCullFace(GLenum);
void glDeleteLists(GLuint, GLsizei);
void glDepthFunc(GLenum);
void glDepthMask(GLboolean);
void glDepthRange(GLclampd, GLclampd);
void glDisable(GLenum);
void glDisableClientState(GLenum);
void glDrawArrays(GLenum, GLint, GLsizei);
void glDrawBuffer(GLenum);
void glEdgeFlag(GLboolean);
void glEnable(GLenum);
void glEnableClientState(GLenum);
void glEnd(void);
void glEndList(void);
void glEvalCoord1d(GLdouble);
void glEvalCoord1f(GLfloat);
void glEvalCoord2d(GLdouble, GLdouble);
void glEvalCoord2f(GLfloat, GLfloat);
void glEvalMesh1(GLenum, GLint, GLint);
void glEvalMesh2(GLenum, GLint, GLint, GLint, GLint);
void glEvalPoint1(GLint);
void glEvalPoint2(GLint, GLint);
void glFinish(void);
void glFlush(void);
void glFogf(GLenum, GLfloat);
void glFogi(GLenum, GLint);
```

```
void glFrontFace(GLenum);
void glFrustum(GLdouble, GLdouble, GLdouble, GLdouble, GLdouble, GLdouble);
GLuint glGenLists(GLsizei);
GLenum glGetError(void);
void glHint(GLenum, GLenum);
void glHistogram(GLenum, GLsizei, GLenum, GLboolean);
void glIndexMask(GLuint);
void glIndexd(GLdouble);
void glIndexf(GLfloat);
void glIndexi(GLint);
void glIndexs(GLshort);
void glIndexub(GLubyte);
void glInitNames(void);
void glLightModelf(GLenum, GLfloat);
void glLightModeli(GLenum, GLint);
void glLightf(GLenum, GLenum, GLfloat);
void glLighti(GLenum, GLenum, GLint);
void glLineStipple(GLint, GLushort);
void glLineWidth(GLfloat);
void glListBase(GLuint);
void glLoadIdentity(void);
void glLoadName(GLuint);
void glLogicOp(GLenum);
void glMapGrid1d(GLint, GLdouble, GLdouble);
void glMapGrid1f(GLint, GLfloat, GLfloat);
void glMapGrid2d(GLint, GLdouble, GLdouble, GLint, GLdouble, GLdouble);
void glMapGrid2f(GLint, GLfloat, GLfloat, GLint, GLfloat, GLfloat);
void glMaterialf(GLenum, GLenum, GLfloat);
void glMateriali(GLenum, GLenum, GLint);
void glMatrixMode(GLenum);
void glMinmax(GLenum, GLenum, GLboolean);
void glMultiTexCoord1d(GLenum, GLdouble);
void glMultiTexCoord1dARB(GLenum, GLdouble);
void glMultiTexCoord1f(GLenum, GLfloat);
void glMultiTexCoord1fARB(GLenum, GLfloat);
void glMultiTexCoord1i(GLenum, GLint);
void glMultiTexCoord1iARB(GLenum, GLint);
void glMultiTexCoord1s(GLenum, GLshort);
void glMultiTexCoord1sARB(GLenum, GLshort);
void glMultiTexCoord2d(GLenum, GLdouble, GLdouble);
void glMultiTexCoord2dARB(GLenum, GLdouble, GLdouble);
void glMultiTexCoord2f(GLenum, GLfloat, GLfloat);
void glMultiTexCoord2fARB(GLenum, GLfloat, GLfloat);
void glMultiTexCoord2i(GLenum, GLint, GLint);
void glMultiTexCoord2iARB(GLenum, GLint, GLint);
void glMultiTexCoord2s(GLenum, GLshort, GLshort);
void glMultiTexCoord2sARB(GLenum, GLshort, GLshort);
void glMultiTexCoord3d(GLenum, GLdouble, GLdouble, GLdouble);
void glMultiTexCoord3dARB(GLenum, GLdouble, GLdouble, GLdouble);
void glMultiTexCoord3f(GLenum, GLfloat, GLfloat, GLfloat);
```

```
void glMultiTexCoord3fARB(GLenum, GLfloat, GLfloat, GLfloat);
void glMultiTexCoord3i(GLenum, GLint, GLint, GLint);
void glMultiTexCoord3iARB(GLenum, GLint, GLint, GLint);
void glMultiTexCoord3s(GLenum, GLshort, GLshort, GLshort);
void glMultiTexCoord3sARB(GLenum, GLshort, GLshort, GLshort);
void glMultiTexCoord4d(GLenum, GLdouble, GLdouble, GLdouble, GLdouble);
void glMultiTexCoord4dARB(GLenum, GLdouble, GLdouble, GLdouble, GLdouble);
void glMultiTexCoord4f(GLenum, GLfloat, GLfloat, GLfloat, GLfloat);
void glMultiTexCoord4fARB(GLenum, GLfloat, GLfloat, GLfloat, GLfloat);
void glMultiTexCoord4i(GLenum, GLint, GLint, GLint, GLint);
void glMultiTexCoord4iARB(GLenum, GLint, GLint, GLint, GLint);
void glMultiTexCoord4s(GLenum, GLshort, GLshort, GLshort, GLshort);
void glMultiTexCoord4sARB(GLenum, GLshort, GLshort, GLshort, GLshort);
void glNewList(GLuint, GLenum);
void glNormal3b(GLbyte, GLbyte, GLbyte);
void glNormal3d(GLdouble, GLdouble, GLdouble);
void glNormal3f(GLfloat, GLfloat, GLfloat);
void glNormal3i(GLint, GLint, GLint);
void glNormal3s(GLshort, GLshort, GLshort);
void glOrtho(GLdouble, GLdouble, GLdouble, GLdouble, GLdouble, GLdouble);
void glPassThrough(GLfloat);
void glPixelStoref(GLenum, GLfloat);
void glPixelStorei(GLenum, GLint);
void glPixelTransferf(GLenum, GLfloat);
void glPixelTransferi(GLenum, GLint);
void glPixelZoom(GLfloat, GLfloat);
void glPointSize(GLfloat);
void glPolygonMode(GLenum, GLenum);
void glPolygonOffset(GLfloat, GLfloat);
void glPopAttrib(void);
void glPopClientAttrib(void);
void glPopMatrix(void);
void glPopName(void);
void glPushAttrib(GLbitfield);
void glPushClientAttrib(GLbitfield);
void glPushMatrix(void);
void glPushName(GLuint);
void glRasterPos2d(GLdouble, GLdouble);
void glRasterPos2f(GLfloat, GLfloat);
void glRasterPos2i(GLint, GLint);
void glRasterPos2s(GLshort, GLshort);
void glRasterPos3d(GLdouble, GLdouble, GLdouble);
void glRasterPos3f(GLfloat, GLfloat, GLfloat);
void glRasterPos3i(GLint, GLint, GLint);
void glRasterPos3s(GLshort, GLshort, GLshort);
void glRasterPos4d(GLdouble, GLdouble, GLdouble, GLdouble);
void glRasterPos4f(GLfloat, GLfloat, GLfloat, GLfloat);
void glRasterPos4i(GLint, GLint, GLint, GLint);
void glRasterPos4s(GLshort, GLshort, GLshort, GLshort);
void glReadBuffer(GLenum);
```

```
void glRectd(GLdouble, GLdouble, GLdouble, GLdouble);
void glRectf(GLfloat, GLfloat, GLfloat, GLfloat);
void glRecti(GLint, GLint, GLint, GLint);
void glRects(GLshort, GLshort, GLshort, GLshort);
GLint glRenderMode(GLenum);
void glResetHistogram(GLenum);
void glResetMinmax(GLenum);
void glRotated(GLdouble, GLdouble, GLdouble, GLdouble);
void glRotatef(GLfloat, GLfloat, GLfloat, GLfloat);
void glSampleCoverage(GLclampf, GLboolean);
void glScaled(GLdouble, GLdouble, GLdouble);
void glScalef(GLfloat, GLfloat, GLfloat);
void glScissor(GLint, GLint, GLsizei, GLsizei);
void glShadeModel(GLenum);
void glStencilFunc(GLenum, GLint, GLuint);
void glStencilMask(GLuint);
void glStencilOp(GLenum, GLenum, GLenum);
void glTexCoord1d(GLdouble);
void glTexCoord1f(GLfloat);
void glTexCoord1i(GLint);
void glTexCoord1s(GLshort);
void glTexCoord2d(GLdouble, GLdouble);
void glTexCoord2f(GLfloat, GLfloat);
void glTexCoord2i(GLint, GLint);
void glTexCoord2s(GLshort, GLshort);
void glTexCoord3d(GLdouble, GLdouble, GLdouble);
void glTexCoord3f(GLfloat, GLfloat, GLfloat);
void glTexCoord3i(GLint, GLint, GLint);
void glTexCoord3s(GLshort, GLshort, GLshort);
void glTexCoord4d(GLdouble, GLdouble, GLdouble, GLdouble);
void glTexCoord4f(GLfloat, GLfloat, GLfloat, GLfloat);
void glTexCoord4i(GLint, GLint, GLint, GLint);
void glTexCoord4s(GLshort, GLshort, GLshort, GLshort);
void glTexEnvf(GLenum, GLenum, GLfloat);
void glTexEnvi(GLenum, GLenum, GLint);
void glTexGend(GLenum, GLenum, GLdouble);
void glTexGenf(GLenum, GLenum, GLfloat);
void glTexGeni(GLenum, GLenum, GLint);
void glTexParameterf(GLenum, GLenum, GLfloat);
void glTexParameteri(GLenum, GLenum, GLint);
void glTranslated(GLdouble, GLdouble, GLdouble);
void glTranslatef(GLfloat, GLfloat, GLfloat);
void glVertex2d(GLdouble, GLdouble);
void glVertex2f(GLfloat, GLfloat);
void glVertex2i(GLint, GLint);
void glVertex2s(GLshort, GLshort);
void glVertex3d(GLdouble, GLdouble, GLdouble);
void glVertex3f(GLfloat, GLfloat, GLfloat);
void glVertex3i(GLint, GLint, GLint);
void glVertex3s(GLshort, GLshort, GLshort);
```

```
void glVertex4d(GLdouble, GLdouble, GLdouble, GLdouble);
void glVertex4f(GLfloat, GLfloat, GLfloat, GLfloat);
void glVertex4i(GLint, GLint, GLint, GLint);
void glVertex4s(GLshort, GLshort, GLshort, GLshort);
void glViewport(GLint, GLint, GLsizei, GLsizei);
```

# Chapter 22

# Module "kde"

A module for the KDE/Qt bindings

This module simply loads the qt module with additional support for the KDE API. It mus be loaded before the "qt" module.

# Chapter 23

# Module "multimouse"

This module provides a simple way to access the Linux /dev/input/mouse* devices. It has been written for multiplayer games where every player has his own mouse.

## 23.1 builtin multimouse_update();

This function updates the data stored in the multimouse namespace.

## 23.2 namespace multimouse

This namespace holds the status information of all mice. It is an array of hashes with one array elements per mouse and one hash entry per property.

E.g. the variable 'multimouse[3].b1' has the value '1' if the left (first) mouse button of the 4th mouse (the array starts with index 0) is pressed.

The following properies are implemented:

```
b1, b2
      Left and right mouse button

dx, dy
      X- and Y-delta for all mousemovement between the last
      call to multimouse_update() and the call before the
      last one.

x, y
      X- and Y-coordinates of the current mouse position.

mx, my
      The maximum X- and Y-coordinates. This variables must
      be set before the variables x and y can be used.
```

```
ob1, ob2, ox, oy
      The old values of b1, b2, x and y before the last call to
      multimouse_update().

direction
      The view-point of the player. Possible values are 'bottom',
      'top', 'left' and 'right'. The default value is 'bottom'.

error
      This variable is defined when an error occured while opening
      the mouse device and is set to the error string.
```

The 'error' variable is read-only. All the other variables are read- and writeable.

# Chapter 24

# Module ”prime”

Example Module for hosted namespaces

## 24.1 namespace prime

This is an example hosted namespace. Numeric keys which are prime do point to true values, other numerical keys point to false values and non-numerical keys are undeclared. Example:

```
debug "declared prime['foo'] = " ~ declared prime['foo'];
debug "prime[5] = " ~ prime[5];
debug "prime[6] = " ~ prime[6];
```

This is just an example for writing modules which provide hosted namespaces. It is not really meant to be used in any applications.

# Chapter 25

# Module "qt"

The SPL/Qt Module

This module provides a thin wrapper for the Qt toolkit. It is using the SMOKE library from the kdebindings package.

## 25.1 manual SPL_Qt;

The SPL/Qt Module

This module provides a thin wrapper for the Qt toolkit. It is using the SMOKE library from the kdebindings package.

All Qt classes are available via the qt namespace and can be instanciated using the "new" keyword, like normal SPL objects:

```
load "qt";

var a = new qt.QApplication();
var l = new qt.QLabel(undef);
l.setText("Hello World!");
a.setMainWidget(l);
l.show();
a.exec();
```

Qt objects can be used like normal SPL objects. However: Only the qt methods are available from SPL. The object properties can't be accessed directly.

The qt.QApplication class has a non-standard constructor which does not take any arguments. It is recommended to use this constructor for creating a qt.QApplication instance.

Some data types can't be converted by this module. So some Qt methods might be unaccesable thru this module.

This is a thin wrapper for the Qt C++ libraries. It is possible to produce segmentation faults or other errors by using this module incorrectly.

This module does not allow you to straight-forward derive your own classes from qt classes. However, this functionality might be added later.

All Qt Objects have a pseudo member-variable ".class" which contains the name of the Qt class for this object.

The pseudo member-variable ".ptr" contains a text representation of the memory address of Qt Object. This can be used e.g. for checking if two SPL Qt Object handlers point to the same Qt Object.

Static methods can be called directly from the qt namespace without instanciating the class. Enums behave like static methods without arguments which do return the numeric value for the enum.

The complete Qt Reference Documentation can be found at: http://doc.trolltech.com/

An SPL Virtual Machine which loaded the "qt" module is unable to dump its state. So the SPL dump/restore feature doesn't work for SPL/Qt programs.

## 25.2 builtin qt_debug(mode);

Enable/disable debug output.

Call this function with an argument of '1' to enable the debug output and with an argument of '0' to disable it.

The debug output is disabled per default.

## 25.3 builtin qt_kde();

Check for kde libraries.

This function returns '1' if the kde APIs are available and '0' if only the core Qt APIs can be used.

## 25.4 builtin qt_ui(ui_file);

Instanciates the widget described in the specified .ui file. The new widget is returned.

## 25.5 builtin qt_child(parent, name, class, recursive);

This is a frontend to the Qt QObject::child() function. The difference is that this function automatically casts the return value to the specified class. Example given:

```
var d = qt_ui("dialog.ui");
var b = qt_child(d, "pushButton1", "QPushButton", 1);
b.setText("Click Me!");
```

## 25.6 builtin qt_cast(object, type);

Cast a Qt Object. The new object handler is returned. For QObjects this is a frontend to QObject::qt_cast().

## 25.7 builtin qt_destroy(object);

Destroy a QObject.

Usually QObjects are destroyed automatically when the last SPL variable refering to it is destroyed and the QObjects has no parent objects.

(Destroying QApplication Objects is automatically delayed until all other Qt Objects are destroyed.)

But sometimes it is neccessary to explicitely destroy a QObject, e.g. when one want's to remove a Widget from a dialog.

## 25.8 builtin qt_delete(object);

Delete a Qt Object.

This function can be used to delete any Qt Object. It is recommended to use qt_destroy() for QObject. This function should only be used for deleting Qt Objects which are not QObjects.

## 25.9 builtin qt_autodelete(object);

Mark a Qt Object for autodeletion.

This maks an object for autodeletion. That means that the object will be deleted automatically when the SPL handler is removed by the garbage collector.

The object is returned by the function. So this function can be used like a filter when instanciating a new object. Example Given:

```
var background = qt_autodelete(new qt.QColor(200, 100, 100));
```

## 25.10 builtin qt_as(type, value);

Create a hint for the SPL/Qt type converter

The SPL/Qt module does its best to guess which version of a method should be used. But sometimes it needs a hint to make the right decision.

E.g. there are two implementations of QMessageBox::question():

```
int QMessageBox::question(QWidget *parent,
            const QString &caption,      const QString &text,
            int button0, int button1 = 0, int button2 = 0)
```

and

```
int QMessageBox::question (QWidget *parent,
            const QString &caption, const QString &text,
            const QString &button0Text = QString::null,
            const QString &button1Text = QString::null,
            const QString &button2Text = QString::null,
            int defaultButtonNumber = 0,
            int escapeButtonNumber = -1)
```

To choose the first version one needs to use qt_as() for the integer parameters so they not converted to strings:

```
qt.QMessageBox.question(win, "Hey!", "Are you sure?",
            qt_as("int", 3), qt_as("int", 4));
```

## 25.11 builtin qt_connect(sender, signal, receiver, slot);

Connect a signal to a slot. This is a frontend to the QObject::connect method. The SIGNAL() and SLOT() macros are not available in SPL. This function simply expects the signal or slot as string. E.g.:

```
qt_connect(mybutton, "clicked()", myapp, "quit()");
```

## 25.12 builtin qt_disconnect(sender, signal, receiver, slot);

Like qt_connect(), but for disconnecting a signal and a slot.

## 25.13 builtin qt_event_callback(object, callback, @eventtypes);

Register an event callback in a Qt object.

The callback function will be passed the QEvent object as 1st parameter. The callback must return true (non-zero) if the event has been comsumed and false (zero) if the event should be passed to the other event handlers.

A list of event types may be passed as additional parameters. If no types are specified, all events are passed thru the callback function.

Example given:

```
function click_callback(e) {
      debug "Click: ${e.x()} / ${e.y()}";
      return 1;
```

```
}

qt_event_callback(widget_object, click_callback,
      qt.QEvent.MouseButtonPress(), qt.QEvent.MouseButtonDblClick());
```

Once a callback function is registered it stays active until the widget gets destroyed. There is no function for unregistering an event callback.

This function can be used to connect an SPL closoure to a Qt widget. The Qt C++ API does not have such a function because C++ has not closoures..

In some cases it is dangerous to copy to variables passed to an event callback to a different context and use it after the callback has been returned. So it is recommended to not do this.

## 25.14 builtin qt_signal_callback(object, signalspec, callback);

Register a signal callback in a Qt object. Example given:

```
function printpos(x,y) {
      debug "New content position: $x / $y";
}

qt_signal_callback(my_scrollview, "contentsMoving(int,int)", printpos);
```

Once a callback function is registered it stays active until the widget gets destroyed. There is no function for unregistering a signal callback.

This function can be used to connect an SPL closoure to a Qt widget. The Qt C++ API does not have such a function because C++ has not closoures..

In some cases it is dangerous to copy to variables passed to a signal callback to a different context and use it after the callback has been returned. So it is recommended to not do this.

## 25.15 builtin qt_virtual_callback(object, method, callback);

Overload a virtual method in a Qt object. The method name is specified in the SMOKE syntax (method names, optionally postfixed with '$', '#' and '?' for scalar, QObject or other arguments).

## 25.16 builtin qt_info();

Return information about classes and methods.

This function returns a data structure describing all classes and methods available thru this interface.

## 25.17 function qt_kdeinit(progname, desc, version);

This function is a simple frontend to KCmdLineArgs::init(). It must be executed before instanciating a KApplication object.

## 25.18 namespace qt

All Qt classes are available via this namespace.

## 25.19 object QtEx

An instance of this object is thrown on internal errors of the Qt wrapper.

### 25.19.1 var QtEx.description;

A description text describing the error.

# Chapter 26

# Module "sdl"

A module for doing multimedia using the SDL library.

Note: Only one SDL window can be opened by one host program at a time. This is a limitation from SDL.

## 26.1 builtin sdl_init(width, height);

Initialize SDL and open a window with the specified size.

The window is opened in fullscreen when the named parameter 'fullscreen' is passed and has a 'true' (non-zero) value.

The window is double-buffered when the named parameter 'doublebf' is passed and has a 'true' (non-zero) value.

## 26.2 builtin sdl_quit();

Terminate the SDL context and close the window.

## 26.3 builtin sdl_title(title, icon);

Set the window and icon name.

## 26.4 builtin sdl_delay(ms);

Delay program execution for the specified number of milliseconds.

Note that the given number of ms is relative to the end of the last call to sdl_delay(). The return code is the effective number of ms the program execution has been delayed.

A zero or negative return code means that the function did not sleep and you are likely to have a timing/performance problem in your application.

## 26.5 builtin sdl_flip();

Update the window (switch backend framebuffers if the window is created in doublebuffering mode).

## 26.6 builtin sdl_update(x, y, w, h);

Update the specified area of the window.

## 26.7 builtin sdl_image_load(filename);

Load the image from the specified filename and return the image handler.

## 26.8 builtin sdl_image_create(w, h);

Create an empty image with the specified size.

## 26.9 builtin sdl_blit(dstimg, srcimg, x, y);

Blit the source image to the specified coordinates in the destination image. If 'undef' is passed as destination image, the image is blitted to the window and will be displayed after the next call to sdl_flip().

## 26.10 builtin sdl_blitrect(dstimg, srcimg, dest_x, dest_y, src_x, src_y, w, h);

Like sdl_blit(), but only blit the specified rectangle.

## 26.11 builtin sdl_copy(srcimg, x, y, w, h);

Copy the the specified rectangle to a new image. The new image is returned.

## 26.12 builtin sdl_fill(dstimg, x, y, w, h, r, g, b, a);

Fill the specified rectangle in the destination image with the specified RGBA values. (The RGBA values are in the range from 0 to 255.)

## 26.13 builtin sdl_fill_pattern(dstimg, x, y, w, h, patternimg);

Fill the specified rectangle in the destination image with a pattern.

## 26.14 builtin sdl_keystat(keyname);

Return '1' if the specified key is pressed and '0' if it isn't. The keynames are listed at:

```
http://www.libsdl.org/cgi/docwiki.cgi/SDLKey
```

(lowercase versions of the SDLK_* strings without the SDLK_ prefix and with blanks instead of underscores)

## 26.15 builtin sdl_sprite_create();

Usually, when doing SDL programming in C, everyone is creating his own SDL sprite library. This is possible when doing SDL programming with SDL to. But for performance reasons I recommend to use the built-in sprites of this module.

This function returns a new sprite object. Such a sprite object has the following attributes:

```
.x
      x-coordinate of the sprite (upper left corner)
.y
      y-coordinate of the sprite (upper left corner)

.z
      z-coordinate of the sprite.
      sprites with higher values overlap sprites with lower values

.image
      the image data for the sprite.
      this must be set to one of SDL Image the object such as
      returned by sdl_image_create().
```

## 26.16 builtin sdl_sprite_redraw();

This redraws all sprites, also those which haven't been updated.

This function is only needed if you are mixing sprites with direct blitting to the window. Usually sdl_sprite_update() is used instead.

In some corner cases (when many sprites have been updated) this function might be faster than sdl_sprite_update(). But this is a very seldom case.

## 26.17 builtin sdl_sprite_update();

This redraws all the updated sprites.

## 26.18 object SdlEx

An instance of this object is thrown on SDL errors.

### 26.18.1 var SdlEx.description;

A description text describing the error.

# Chapter 27

# Module "sql"

The base module for accessing SQL databases.

Additional database driver modules need to be loaded for the backend databases.

## 27.1 builtin encode_sql(text);

This function encodes a string to be used in an SQL query. I.e. the string will be put in single quotes and every single quote in it will be replaced with two single quotes.

This function is designed to be used with the encoding/quoting operator (::).

## 27.2 builtin sql_connect(driver_name, driver_data);

This function creates an SQL database handle.

The backend driver for the database must be loaded already when this function is called. E.g.:

```
load "sql_sqlite";
var db = sql_connect("sqlite", "");
```

The database handler returned by this function may then be used as first argument for the sql() function.

An SqlEx exception is thrown on errors.

## 27.3 builtin sql(database_handler, query);

This function executes an SQL query.

If the query returns data, it is passed back using the return value of this function.

The database handler must be created using sql_connect().

The return value is an array with an element for every database tuple in the result set. Those elements then have a named children for each field in the tuple, with the field name returned from the database as key. E.g.:

```
var db = sql_connect("sqlite", "/var/lib/myapp/database.db");

var r = sql(db, "SELECT username, userid FROM users");

foreach i (r)
      debug "User '${r[i].username}' has ID '${r[i].userid}'.";
```

An SqlEx exception is thrown on errors.

## 27.4 object SqlEx

An instance of this object is thrown on database errors.

### 27.4.1 var SqlEx.description;

A description text describing the error. Some backend drivers might add additional object members.

# Chapter 28

# Module "sql_mysql"

This module implements the MySQL database driver.

Load this module (and the "sql" module) and pass "mysql" as backend driver name to sql_connect();

The string describing the database connection (the 2nd argument to sql_connect()) is a coma seperated list of the following tokens:

```
host={hostname}
user={username}
pass={password}
sock={socket_filename}
db={database_name}
port={tcp_port_number}

compress             (set MySQL CLIENT_COMPRESS flag)
ignore_space         (set MySQL CLIENT_IGNORE_SPACE flag)
multi_statements     (set MySQL CLIENT_MULTI_STATEMENTS flag)
```

Whitespaces are not allowed. E.g.:

```
var db = sql_connect("mysql", "host=dbserver,user=dbuser");

var r = sql(db, "show databases");

foreach i (r)
      debug pop r[i];
```

## 28.1 builtin encode_mysql(text);

This function encodes a string to be used in an MySQL query. I.e. the string will be put in single quotes and special characters inside the string is quoted with backslash sequences.

Always use this function instead of encode_sql() for quoting strings in MySQL queries unless you are running MySQL in the ANSI_QUOTES mode.

This function is designed to be used with the encoding/quoting operator (::).

# Chapter 29

# Module ”sql_postgres”

This module implements the POSTGRES database driver.

Load this module (and the ”sql” module) and pass ”postgres” as backend driver name to sql_connect();

The string describing the database connection (the 2nd argument to sql_connect()) is used as it is as PQconnectdb conninfo string. A detailed description of the format of this string can be found at:

```
http://www.postgresql.org/docs/8.0/interactive/libpq.html#AEN22513
```

Example given:

```
var db = sql_connect("postgres", "host=dbserver user=postgres dbname=template1");

foreach[] tuple (sql(db, "show all"))
      debug "${shift tuple} = ${shift tuple}";
```

# Chapter 30

# Module ”sql_sqlite”

This module implements the SQLite database driver.

Load this module and the ”sql” module and pass ”sqlite” as backend driver name to sql_connect();

# Chapter 31

# Module "sql_utils"

Additional SQL helper functions.

The "sql" module just provides a generic function for doing SQL queries, namely sql(). This module declares some wrapper functions for sql() which do implement common usage scenarios for sql().

## 31.1 function sql_value(db, query);

Execute the query and return the first field of the first tuple in the result.

## 31.2 function sql_tuple(db, query);

Execute the query and return the first tuple in the result.

## 31.3 function sql_keyval(db, query);

Execute the query, which must return fields named 'key' and 'value', and return a hash created from this result set.

## 31.4 function sql_array(db, query);

Execute the query, which must return fields named 'value', and return an array created from this result set.

# Chapter 32

# Module "system"

This module provides a simple API for calling shell commands

## 32.1 builtin system(command, encoding);

This function calls the specified shell command and returns the text written to stdout.

If the output is not UTF-8 encoded, the encoding must be specified with a 2nd parameter. Valid encodings are the same as for the '#encoding' compiler pragma (see 'SPL Language Reference' for a list).

If UTF-8 encoding is expected but the output fails to pass the UTF-8 test, the output is assumed to be latin1 encoded.

A newline at the end of the output is automatically removed.

# Chapter 33

# Module "task"

SPL Task Management Module

This module provides basic functions for handling SPL tasks. SPL tasks may be used like threads, co-routines, or anything simmilar.

## 33.1 builtin task_create(name, code, ctx);

This function creates a new task. The 1st parameter is the name of the new task and the 2nd is the SPL program code for the task. The 2nd parameter will be compiled when executing the function, so it is a good idea to keep it small and move the big portion of the task logic to seperate functions. The third option is the context in which the task should be executed.

If the task name is undefined, the task won't have a name attached to it and it won't be possible to access it later.

If the context is ommitted, the task will run in the same context as the function which called this function.

## 33.2 builtin task_pause(name);

This functions pauses the specified task until it is woken up by task_continue() (or by an external event).

If the task name is ommitted, the current task will be affected by this function.

## 33.3 builtin task_continue(name);

This function wakes up the specified task. It also can be used to remove the SYSTEM flag from the current tast (see task_system()).

If the task name is ommitted, the current task will be affected by this function.

## 33.4 builtin task_system(name);

This function sets the SYSTEM flag on the specified task. That means that the scheduler will not schedule any other task until the system flag is removed again using task_continue() or the task is paused (task_pause()) or killed (task_kill()).

If the task name is ommitted, the current task will be affected by this function. Usually this function is only used to manipulate the currently running task.

## 33.5 builtin task_public(name);

This function is used to set the PUBLIC flag on a task. This is e.g. needed when the task should be woken up by WebSPL when the taskname is encoded into the session id.

If the task name is ommitted, the current task will be affected by this function.

## 33.6 builtin task_getname();

Returns the name of the current task.

## 33.7 builtin task_check(name);

Returns 1 if the specified task exists and 0 otherwise.

## 33.8 function task_kill(name);

This function kills the specified task.

If the task name is ommitted, the current task will be affected by this function.

## 33.9 function task_late_kill(name);

This function schedules the specified task to be killed. The task will be killed when becomes paused the next time (e.g. because task_pause() is called on it). If the task is already paused, it will be killed emediatly.

If the task name is ommitted, the current task will be affected by this function.

## 33.10 function task_switch(name);

This function pauses the current task and wakes up the specified one. It is using task_system() to make the operation atomic and so avoid race conditions.

## 33.11 function task_co_call(name);

This function is like, task_switch() but registers the current task as co-routine caller. That means, when the specified task call task_co_return(), the calling task will be woken up again and the parameter to task_co_return() will be returned by this function.

## 33.12 function task_co_return(retval);

Return to the task which switched to this task using task_co_call(). If the current task was not entered using this mechanism, control is passed to the task specified by task_co_setdefault(), or a runtime error is triggered if no default has been defined.

## 33.13 function task_co_setdefault(name, def);

Set the task "def" as default task to be returned when the task "name" calls task_co_return() was not entered thru task_co_call().

## 33.14 function task_eval(code, ctx);

An eval, implemented as own task. It is recommended to use the "eval" statement instead. The only difference is that this function allows you to specify a different function context in which the code should be executed.

# Chapter 34

# Module ”termio”

Terminal IO module

This module provides simple functions for controlling ANSI terminals.

## 34.1 builtin termio_setpos(x, y);

This function sets the current curos position The X and Y coordinates are relative to the upper left corner. The upper left corner has the coordinates (1, 1).

## 34.2 builtin termio_clear();

This function clears the screen.

## 34.3 builting termio_sleep();

This function sleeps for one second.

# Chapter 35

# Module "time"

A module for time and time manipulations

## 35.1 builtin time();

This function returns the number of seconds since the epoch.

## 35.2 builtin time_local(time);

This function converts the number of seconds since the epoch (as returned by time()) so a node with the childs .sec, .min, .hour, .mday .month, .wday, .year, .yday, .gmtoff, .isdst and .zone. (It is simply a wrapper to the struct tm as defined in time.h.)

## 35.3 builtin time_gm(time);

This function converts the nuber of seconds since the epoch (as treturned by time()) to a time node as returned by time_local(), but in GMT.

## 35.4 builtin time_diff(time1, time0);

This function returns the difference in seconds between time1 and time0. It expects two ints containing the number of seconds since the epoch as returned by time().

## 35.5 builtin time_fmt(format, tm);

This Function formats a time node as returned by time_local() according to the strftime manpage. This is simply a wrapper to the strftime() C library function.

## 35.6 builtin time_mk(tm);

This function converts a time node as returned by time_local() to the count of seconds since the epoch as returned by time().

## 35.7 builtin time_mkgm(tm);

This function converts a time node as returned by time_gm() to the count of seconds since the epoch as returned by time(), it works similar to mktime but assumes the time node is in GMT.

This function is not available on the CYGWIN platform.

# Chapter 36

# Module "uuidgen"

This module provides a simple API for creating UUID strings

## 36.1 builtin uuidgen();

This function generates a UUID (using libuuid) and returns a 36 character string with a hex representation of th uuid.

# Chapter 37

# Module "w2t"

The SPL Web 2.0 Toolkit

This module is EXPERIMENTAL and not so well document yet. Expect more to come..

## 37.1 var w2t_core_scripts = <>;

The core w2t browser-side scripts.

This JavaScript code block defines the following functions:

```
function w2t_encode_xml(text)

      XML-encode the parameter and return the encoded text

function w2t_console_open()

      Open a W2T console window (for debug purposes).
      Usually one adds a little button or link to the development
      versions of an application which calls this function.

function w2t_console(title, text)

      This function can be used to write a message to all open
      W2T console windows.

function w2t_proccess_dom(n)
function w2t_proccess(xmldata)
function w2t_update_status()
function w2t_gc()

      Some functions used internally by w2t_send() for creating the
      XML requests and processing the XML responses.

function w2t_send(xmldata, onfinish)
```

```
      Send a request specified in 'xmldata' to the server and call
      'onfinish' after the response has been processed. Usually
      one is calling w2t_callback() instead of using w2t_send()
      directly.

function w2t_callback(callback_name, arguments)

      Call a callback on the server.
      The second argument is an array of arguments.

function w2t_animate(finish, workers, seconds, fps)

      A generic helper function for animations. The animation runs
      for the specified number os seconds with the specified number
      of frames per second. Frames are automatically skipped when
      rendering becomes to slow. For each frame all functions in
      the 'workers' array are called with a floating point value
      as parameter (0 for the first frame up to 1 for the last one).

function w2t_animate_fade(id, attr, newval)
function w2t_animate_svgmove(id, newx, newy)

      Helper functions which do return function pointers to
      dynamically created worker functions for w2t_animate().
```

## 37.2 object W2t

The W2t object.

### 37.2.1 static W2t.scripts = [ w2t_core_scripts ];

An array with JavaScript fragments. Other modules can push their own script fragments to this array when loaded. This scripts are then automatically included in the application startup page.

### 37.2.2 static W2t.styles = [ ];

An array with CSS stylesheet fragments. Other modules can push their own css fragments to this array when loaded. This scripts are then automatically included in the application startup page.

### 37.2.3 static W2t.w2t_ns = "http://www.spl-scripting.org/xml/w2t";

The namespace URI for the "w2t" namespace prefix.

### 37.2.4 static W2t.xmlns;

A hash of namespace declarations one usually wants to use for Web 2.0 applications (prefix as key and URI as value). Other modules (and w2t applications) may add additional namespaces. Per default the following namespaces are defined:

```
w2t     w2t_ns
svg     http://www.w3.org/2000/svg
xhtml   http://www.w3.org/1999/xhtml
xlink   http://www.w3.org/1999/xlink
```

This hash is used by getxmlnsdecls().

### 37.2.5 var W2t.startup;

The startup page content. This variable must be set before calling run().

### 37.2.6 var W2t.mime_type = "text/xml";

The mime type of the startup page.

### 37.2.7 var W2t.callbacks;

A hash with all callbacks.

Use register_callback() to add entries.

### 37.2.8 var W2t.callbacks_oncleanup;

A hash with oncleanup event handlers for the callbacks. When the callback garbage collector cleans up a callback, it also executes the oncleanup handler if there is one.

Use register_callback() to add entries.

### 37.2.9 var W2t.callbacks_rev = 0;

A revision counter for the callbacks list. It is automatically incremented by register_callback(). This counter is needed to work arround possible race conditions in the client-side garbage collection algorithm.

### 37.2.10 var W2t.request;

The current request (when run() is active). This is an XML document handler.

### 37.2.11 var W2t.response;

The current response (when run() is active). This is an XML document handler.

### 37.2.12 method W2t.register_callback(id, handler, oncleanup);

Register a new callback.

### 37.2.13 method W2t.response_add(xmltext);

Add a new XML snippet to the response document.

### 37.2.14 method W2t.dom_appendchild(id, xmltext);

Append a child node in the client DOM tree.

### 37.2.15 method W2t.dom_remove(id);

Remove a node from the client DOM tree.

### 37.2.16 method W2t.dom_replaceinner(id, xmltext);

Replace the inner XML text of node in the client DOM tree.

### 37.2.17 method W2t.dom_setattr(id, name, value);

Set (and create if needed) an attribute.

### 37.2.18 method W2t.execute(script);

Execute the javascript fragment passed as argument in the client.

### 37.2.19 method W2t.getscripts();

Return the concatenated contents of scripts with an additional header. This must be included in a <script><![CDATA[ .. ]]></script> block within the startup document.

### 37.2.20 method W2t.getstyles();

Return the concatenated contents of styles. This must be included in a <style type="text/css"><![CDATA[ .. ]]></style> block within the startup document.

### 37.2.21 method W2t.getxmlnsdecls();

Return a string with 'xmlns:PREFIX="URI"' xml namespace declarations built using the xmlns hash described above.

### 37.2.22 method W2t.getid();

Get a unique id, e.g. for the id="" attribute of auto-generated XML elements and for callback names.

### 37.2.23 method W2t.run();

Main runloop. This function never returns. It just calls prior registered callback functions.

### 37.2.24 method W2t.init();

Constructor.

# Chapter 38

# Module "webdebug"

A module that implements an SPL debugger that can be used with a Web Interface.

Only load that module when you are in a save test environment do not let end-users access the debugger frontend!

## 38.1 function webdebug() ;

Create a Web Deugger task and return the URI for it. E.g.:

```
<script><!--
window.open("${webdebug()}", "",
      "width=600,height=500,resizable=yes,scrollbars=yes");
//--></script>
```

# Chapter 39

# Module "wsf"

WebSPL Forms Library

This module provides a frame work for doing application development with WebSPL. The basic idea is to split down the user interface into so-called components. Each component provides a part of the DOM tree which is displayed in the browser window. There is a seperate program task for each component - so they can act as independent as the programmer wants.

This module provides WsfComponent, the base object for WSF Components, and WsfDocument, the object which manages the interaction with the Browser.

Other Modules provide additional WSF Components. E.g.:

```
WsfDebug      WsfDialog
WsfDisplay      WsfEdit
WsfEditSql      WsfGraph
WsfMenu            WsfSwitch
```

If the browser has support for it, WsfDocument does only send those parts of the DOM tree to the browser which have actually changed and replace them 'in place' in the current page using a little JavaScript hack.

## 39.1 function wsf_get_domupdate_status(@methods);

Checks the HTTP Agent string and auto-detects if the browser is able to synamically update the DOM tree.

It is possible to specify methods as arguments in the order of preference. Then this methods are checked in that order. E.g.:

```
var method = wsf_get_domupdate_status("iframe", "xmlhttprequest");
```

Will return "iframe", "xmlhttprequest" or "none".

If no arguments are given the function eighter returns "iframe" or "none". The "xmlhttprequest" method is left out in this case.

Usually this is only used internally by the WsfDocument object to initialize WsfDocument.domupdate and doesn't need to be called by the user.

## 39.2 object WsfComponent

The base WsfComponent object. All other Wsf Components are derived from this Object.

### 39.2.1 static WsfComponent.id_counter = 0;

This counter is used by the constructor to create the id for new instances of Wsf Components.

### 39.2.2 var WsfComponent.id;

The id of this component instance. It is set automatically by the contructor and must be included as "id" attribute in the top element of the HTML tree returned by get_html().

This also is the name of the task created for this component by the constructor.

### 39.2.3 var WsfComponent.sid;

The sid (session id) for the task running this component. It must be included as parameter 'sid' in all query strings. The methods add_action(), add_href() and add_javascript() should be used for creating links, etc.

### 39.2.4 var WsfComponent.dirty = 0;

This variable must be set to 1 whenever it is neccessary to call get_html() again and refresh the browser view of this component.

It is also possible to set this to 1 if there have been any changes in the children array.

### 39.2.5 var WsfComponent.children;

An array of child components. The keys may be freely choosen. The get_html() method must run get_html() for all children and include the return value in its own output.

### 39.2.6 var WsfComponent.main_task;

The name of the main task running the WsfDocument instance which is responsible for this component. This is automatically set by WsfDocument when checking for set dirty flags and by the child_set() method.

### 39.2.7 method WsfComponent.add_action(url);

Whenever a form is generated by get_html(), this method must be used to create the "action" attribute in the <form> tag. A hidden input field for the "sid" parameter mus also be generated. E.g.:

```
<form id="$id" ${add_action(cgi.url)}>
<input type="hidden" name="sid" value="${sid}" />
...
</form>
```

### 39.2.8 method WsfComponent.add_href(url);

Whenever a HTML link is created by get_html(), this method must be used to create the "href" attribute in the <a> tag. E.g.:

```
<a ${add_href("${cgi.url}?sid=${sid}&foo=bar")}>Foobar</a>
```

### 39.2.9 method WsfComponent.add_javascript(url);

Whenever JavaScript is used in the code created by get_html(), this method must be used to create the statement which sets location.href to the new value. E.g.:

```
onClick="${ add_javascript("${cgi.url}?sid=${sid}&foo=bar") }"
```

the URL will be quoted with single quotes in the generated code. So there is no problem with embedding it using "onFoobar" JavaScript event handlers. No additional quoting is done. So something like that for creating query string parameters with JavaScript is possible too:

```
onClick="${ add_javascript("${cgi.url}?sid=${sid}&foo=' + (3+5) + '") }"
```

### 39.2.10 method WsfComponent.get_html_children();

A simple method for calling get_html_cached() on all children and concatenating the results.

### 39.2.11 method WsfComponent.get_html_cached();

This method returns the HTML code for this component. If the dirty flag is set, get_html() is called to generate the HTML code. Otherwise a cached version of the HTML code is returned.

### 39.2.12 method WsfComponent.get_html();

The method for creating the HTML code. The top HTML element must have the attribute "id" set to the value of the id member variable. The default behavior is to simply return:

```
'<div id="$id">\n' ~ get_html_children() ~ '</div>\n'
```

### 39.2.13 method WsfComponent.main();

The main function for the task running this component. It is first called by the constructor and is running until it calls task_co_return().

After that, get_html() is called by the WsfDocument object to create the HTML representation.

The task_co_return() function returns when the user has done something in his browser window which effects this component; i.e. has clicked a link or submitted a form generated by get_html().

Then this function can react to the event and call task_co_return() again when it has finished processing this event. Don't forget to set dirty to 1 if get_html() needs to be called again.

If this method does not call task_co_return(), the task will hang in an endless loop. So don't forget to do that!

### 39.2.14 method WsfComponent.child_set(name, obj);

Create (substitute) a child component. The 1st parameter is the key in the children array, the 2nd parameter the new component object.

### 39.2.15 method WsfComponent.child_remove(name);

Remove a child component. The parameter is the key in the children array.

### 39.2.16 method WsfComponent.destroy();

The destructor. It needs to be called when the object isn't needed anymore to kill the task assigned to that object. It also calls the destroy method of all child components.

It is save to call that from the main() method. If you do so, killing the task is postponed until main() calls task_co_return() the next time. main() will then never return from this function again.

### 39.2.17 method WsfComponent.init();

The constructor.

## 39.3 object WsfDocument

The WsfDocument object. There must be one WsfDocument object for every browser window which is under control of WSF. Usually it is instanciated and assigned to a variable called 'page', then initialized and finally the main() method is called. E.g.:

```
var page = new WsfDocument();
page.title = "My Application Title";
page.root = new MyFunnyRootWsfComponent();
page.main();
```

The main() method never returns.

### 39.3.1 static WsfDocument.domupdate = wsf_get_domupdate_status();

This variable is automatically set when the module is loaded. It contains the return code of wsf_get_domupdate_status().

It is possible to change that variable before instanciating WsfDocument the first time. After that, the variable shouldn't be touched anymore.

Additionally it is possible to change that variable by passing a cgi query string parameter on program startup:

```
wsf_domupdate={ iframe | xmlhttprequest | none }
```

If the domupdate mechanism is running in "iframe" mode, the iframe can be set visible by passing the query string parameter "wsf_showiframe".

The "xmlhttprequest" method is probably the best implementation, but right now it is not possible to do file uploads using this method.

### 39.3.2 static WsfDocument.showiframe = 0;

If domupdate is set to "iframe" and this variable is set to 1, the iframe used for the domupdate mechanism will be visible. This is only of interest for debugging purposes.

### 39.3.3 var WsfDocument.root;

This is the root component for this WsfDocument. It must be set to an instance of WsfComponent (or any derived object) before the main() method is called.

### 39.3.4 var WsfDocument.title = "";

The content for the <title> tag generated by this object. This can not be changed later, if domupdate is set to "iframe". So it must be set before calling main(), or not at all.

### 39.3.5 var WsfDocument.html_head = "";

HTML code to be included between <head> and </head>. Must be set before calling main(), or not at all.

### 39.3.6 var WsfDocument.body_attr = "";

HTML attributes to be set in the <body> tag. Must be set before calling main(), or not at all.

### 39.3.7 var WsfDocument.callbacks;

A hash with arrays of callback functions. Use callback_add and callback_del to maintain the entries.

### 39.3.8 method WsfDocument.callback_add(type, func);

Add a callback function to the callbacks data structure.

The following callback types are called by this object:

```
pre_update      called before the html (xml) output is created.
post_update      called after the html (xml) output is created.
```

### 39.3.9 method WsfDocument.callback_del(type, func);

Add a callback function from the callbacks data structure.

### 39.3.10 method WsfDocument.callback_call(type, %options);

Call all callbacks of a type. The callback is passed the type as first parameter and this WfsDocument object as 2nd parameter. The named parameters are also passed thru to the callback functions.

### 39.3.11 method WsfDocument.main();

This method implements the main loop of a WsfDocument. It must be called after setting up the variables described below.

It handles the creation of HTML pages which are then passed to the browser. Whenever the WsfComponent.main() method calls task_co_return() after processing a user event, control is passed back to this method so it can update the browser window.

This function does never return.

# Chapter 40

# Module "wsf_action"

A module which implements a WSF Component with support for dynamically registered callbacks (usually using closures) in the templates.

## 40.1 object WsfAction WsfComponent

A WSF Component with support for dynamically registered callbacks. An example:

```
var mywsfaction = new WsfAction(function() {
      return #file-as-template *demotpl; }, undef);

#embedded-file demotpl EOT
<div id="$id">
      <a ${add_href("${cgi.url}?sid=${sid}&mode=foobar")}>FooBar</a>
      action("foobar", function() { debug "FooBar!!"; });
</div>
EOT
```

Usually the callbacks are created on-demand by a more complex logic and detailed information for the context of the callback is passed using a closure.

This object is derived from WsfComponent.

### 40.1.1 var WsfAction.action_list;

A hash containing the function pointers for the registered actions.

### 40.1.2 var WsfAction.template_function;

The function pointer passed to the constructor.

### 40.1.3 var WsfAction.template_options;

The options hash passed to the constructor.

### 40.1.4 var WsfAction.data;

This variable can be used to save persistent data between calls of template_function.

### 40.1.5 method WsfAction.action_reset();

This method must be executed by get_html() before evaluating the template. It resets action_list.

### 40.1.6 method WsfAction.action(name, callback);

Registeres an action. If the name is set as CGI parameter, the callback is called from main().

When the callback is called, the following named arguments are passed to it:

```
- action:
      the name under which this callback has been registered

- obj:
      this WsfAction instance

- all the options in the template_options variable
```

### 40.1.7 method WsfAction.main();

Overloaded WsfComponent.main().

Calls the registered actions and sets the dirty flag. The calling convention for the actions is described in the section about the action() method above.

### 40.1.8 method WsfAction.get_html();

Overloaded WsfComponent.get_html().

This calls template_function() to generate the html text for this component. The named arguments declared in template_options are passed to template_function(). The function is called in the context of this WsfAction instance.

### 40.1.9 method WsfAction.init(_template_function, %_template_options);

The Constructor.

# Chapter 41

# Module "wsf_debug"

A module that implements an SPL debugger that can be used with a Web Interface and integrates nicely in WSF applications.

## 41.1 object WsfDebug WsfComponent

The WSF debugger compontent.

This WSF component creates a small "controler box" with a button for opening a debugger window and, if the DOM update machnism (see "wsf") is active, add buttons for displaying and hiding the iframe used for the DOM update hack.

Derived from WsfComponent.

### 41.1.1 method WsfDebug.main() ;

Overloaded WsfComponent.main().

### 41.1.2 method WsfDebug.get_html() ;

Overloaded WsfComponent.get_html().

# Chapter 42

# Module "wsf_dialog"

A module which implements a "Qt Designer"-like component for creating WSF dialogs in a web browser.

## 42.1 object WsfDialog WsfComponent

A WSF component for creating simple WSF components in a web browser.

A Dialog is simply a collection of form elements with small code snippets, which are automatically executed when the content of the element has been modified (or the button been pushed).

Such a dialog can be dumped as XML and later loaded again.

If the "save" button should be functional, you eighter need to derive your own object from this one or overwrite the method name after instanciating this object.

Derived from WsfComponent.

### 42.1.1 var WsfDialog.xmltree;

The configuration of the dialog, in the format described in format_xml_parse().

All changes are done directly in this data structrue. So there is no need of calling any kind of convertation function in order to keep this structure up to date.

Simply call format_xml_dump() on this variable to create an XML file for this dialog (usually this is done in an overloaded save() method.

### 42.1.2 var WsfDialog.values;

The current values of the elements. If there is an input box "foobar", "values.foobar" can be used to read and modify the value of that input box.

### 42.1.3 var WsfDialog.edit_mode = 0;

When this variable is set to '1', the dialog is currently in edit mode. Usually applications only allow privelidged users to switch to edit mode.

The dirty variable (WsfComponent.dirty) must be set when this variable is changed. Usually the method set_edit_mode() is used to set this variable.

### 42.1.4 var WsfDialog.edit_current = "";

The name of the element which is open in the editor right now. If this is set to "", the screen with the generic dialog options is displayed.

Usually this is only set internally by main() when the user changes the current element in the web frontend.

### 42.1.5 var WsfDialog.edit_show_xml = 0;

The frontend allows displaying the current XML tree so it can e.g. be copy&pasted to a file. This variable is set to '1' when displaying of the XML tree is currently active.

### 42.1.6 var WsfDialog.show_edit_button = 0;

Set this to '1' if you want this component to add an 'edit' button to the component. With this button, the user can switch to the editor view at any time. This should be used with care since someone with access to the editor may access any SPL code! Only privelidged users show have access to this feature.

### 42.1.7 method WsfDialog.set_edit_mode(newmode);

Call this function if you want to switch this component to edit mode (parameter = '1') or back to executing the dialog (parameter = '0').

### 42.1.8 method WsfDialog.get_html();

Overloaded WsfComponent.get_html().

### 42.1.9 method WsfDialog.save();

This method is called when the user clicks on the "save" button in the dialog editor. It does nothing in the default implementation. So you need to overload it in a derived object to add this functionality.

### 42.1.10 method WsfDialog.main();

Overloaded WsfComponent.main().

### 42.1.11 method WsfDialog.reset_values();

Reset the dialog elements to their default values. This can also be used in the code snippets assigned to the dialog elements. To reset the dialog.

### 42.1.12 method WsfDialog.import_xml(xmldata);

Overwrite xml data. The argument is the XML text. This might also be called by one of the code snippets attached to dialog elements to switch to another dialog. (E.g. when clicking on a "Next" button.)

### 42.1.13 method WsfDialog.init(xmldata);

The constructor.

The parameter is the XML text containing the dialog configuration. If it is undef (or not specified at all), an empty dialog will be created.

# Chapter 43

# Module ”wsf_display”

A module which implements a simple WSF Component for displaying stuff.

## 43.1 object WsfDisplay WsfComponent

A WSF Component for displaying stuff.
It always returns HTML following the schema:

```
<$tag $attributes id="$id">$html_text</$tag>
```

Derived from WsfComponent.

### 43.1.1 var WsfDisplay.tag = ”div”;

The tag name used for the output of this component.

### 43.1.2 var WsfDisplay.attributes;

Additional attributes to be added in the HTML output.

### 43.1.3 var WsfDisplay.html_text;

The inner HTML of this component.

### 43.1.4 method WsfDisplay.get_html();

Overloaded WsfComponent.get_html()

### 43.1.5 method WsfDisplay.set_html(t);

Set html_text and mark this component as dirty to refresh the output.

### 43.1.6 method WsfDisplay.init(t);

Constructor. The html_text variable will be set to the value passed as parameter.

# Chapter 44

# Module "wsf_edit"

A module which provides a generic WSF component for creating edit frontends for any kind of data tuples. See "wsf_edit_sql" for an implementation.

## 44.1 object WsfEdit WsfComponent

This is a base object for WSF components which assemble edit frontends for any kind of data tuples. Usually one would not use this object directly but an object which is derived from it, such as WsfEditSql.

A 'data tuple' may be anything that can be represented as list of key-value pairs, such as a tuple from an SQL table.

The basic idea is that the methods edit_load() and edit_save are overloaded so they can load and save the data tuple to be edited.

The get_html() method should also be overloaded so it evaluates a template which is using the methods edit_input(), edit_textarea and edit_select() for creating the form elements.

An example for using WsfEdit can be found in the documentation of WsfEditSql.

This object is derived from WsfComponent.

### 44.1.1 var WsfEdit.edit_data;

A hash containing the data tuple. When edit_save() is called, this contains the data sent to the browser, not the new data returned by the user. This can be used to check if the data has actually been modified.

### 44.1.2 var WsfEdit.edit_fields;

The fields included in the edit form generated. This can be used to check which fields in the tuple have actually been shown to the user. The edit_save() method should only save the fields defined in this hash. This is especially important if

the template includes some fields conditionally, e.g. because the user has not the accurate permissions to edit all fields.

### 44.1.3 var WsfEdit.edit_actions;

This hash can be used to define additional actions to be executed instead of simply calling edit_save().

If a query string parameter matches a key defined in that hash, the value of that hash entry will be executed (as function/method pointer) and edit_save(). Won't be called.

Usually this is used to implement things such as "Back to List" buttons.

### 44.1.4 var WsfEdit.just_updated;

The edit_save() method should set this variable to '1' when is saved the new data. This can e.g. be used by the template to display a "Database updated" message.

### 44.1.5 method WsfEdit.edit_reset();

This method must be executed by get_html() before evaluating the template. It resets edit_fields and calls edit_load().

### 44.1.6 method WsfEdit.edit_input(field, %opt);

Returns the HTML code for a input box.

The parameter 'field' is a key in the edit_data hash.

The following named parameters are supported:

```
style      value for the css style attribute
size      value for the size attribute
```

Simply include that function in your template using the ${ .. } substitution.

### 44.1.7 method WsfEdit.edit_textarea(field, %opt);

Returns the HTML code for a textarea.

The following named parameters are supported:

```
style      value for the css style attribute
cols      value for the cols attribute
rows      value for the rows attribute
```

(see also edit_input())

### 44.1.8 method WsfEdit.edit_select(field, list, %opt);

Returns the HTML code for a select box.

The list parameter is a hash with the possible values as keys and the description texts as values.

The following named parameters are supported:

```
style      value for the css style attribute

emptydesc
      If this parameter is defined, an additional option with
      that description and an empty value is added as first
      option to the list of possible options.
```

(see also edit_input())

### 44.1.9 method WsfEdit.edit_load();

This method must be overloaded.

It must fill the edit_data hash with values.

### 44.1.10 method WsfEdit.edit_save();

This method must be overloaded.

For all keys defined in edit_fields, it must update the values in the backend storage.

The data must be obtained from 'cgi.param["edit_$keyname"]'.

It is possible to compare the new values with the old values in edit_data to see if the values have actually changed.

The variable just_updated must be set to '1' if anything has been updated.

### 44.1.11 method WsfEdit.main();

Overloaded WsfComponent.main().

### 44.1.12 method WsfEdit.get_html();

Overloaded WsfComponent.get_html().

This method must be overloaded again by object actually implementing an editor. The method edit_reset() must be called in the very beginning of that method.

### 44.1.13 method WsfEdit.init();

Constructor.

# Chapter 45

# Module "wsf_edit_sql"

A module which provides a generic component for editing records in SQL databases, based on WsfEdit.

## 45.1 object WsfEditSql WsfEdit

A WSF Component for editing records in SQL databases.

E.g. there is a database handle 'db' (see "sql") for accessing a database with a table 'user'. In this table exist the fields 'firstname' and 'lastname' which should be editable using this comonent. Records are indexed using the field 'userid'. The field 'username' should be displayed in the form, but is not editable. Creating an editor for record 42 in this table could look like this:

```
var editor = new WsfEditSql(42, db, "user", "userid",
      function() { return #file-as-template *edit_template; });

#embedded-file edit_template END-OF-TEMPLATE
<form id="$id" ${add_action(cgi.url)}>
      Editing user '${edit_data.username}':             <br/>
      Realname:  ${edit_input("", "60", "realname")}    <br/>
      Firstname: ${edit_input("", "60", "firstname")}
</form>
END-OF-TEMPLATE
```

This object is derived from WsfEdit.

### 45.1.1 var WsfEditSql.sql_id;

The ID of the record in the databse. This is looked up in field spl_key.

### 45.1.2 var WsfEditSql.sql_db;

The database handle.

### 45.1.3 var WsfEditSql.sql_table;

The table name.

### 45.1.4 var WsfEditSql.sql_key;

The field which is used to find the record to be edited. (see also spl_id)

### 45.1.5 var WsfEditSql.get_html_core;

The function which evaluates the template. When executed, this will be context-dispatched to this object (using the '*' operator).

### 45.1.6 method WsfEditSql.edit_load();

Overloaded WsfEdit.edit_load().

### 45.1.7 method WsfEditSql.edit_save();

Overloaded WsfEdit.edit_save().

### 45.1.8 method WsfEditSql.get_html();

Overloaded WsfEdit.get_html().

Simply calls get_html_core (context-dispatched to this object) to do the real work.

### 45.1.9 method WsfEditSql.init(_sql_id, _sql_db, _sql_table, _sql_key, _get_html_core);

The Constructor.

The parameters are simply copyied to the variables described above.

See the general discription of this object (WsfEditSql) for a usage example.

# Chapter 46

# Module "wsf_graph"

A module which implemnts a WSF Component for displaying and editing graphs.

## 46.1 object WsfGraph WsfComponent

A WSF Component for displaying and editing graphs.

In order for this component to work properly, the browser must have support for iframes and must allow javascripts to modify the DOM tree. This is the case with most of todays graphical web browsers.

For various technical reasons, the graph editor itself is displayed in an iframe. This object creates an extra SPL task for this iframe.

The constructor is passed a function which returns a list of node objects which must implement the WsfGraphNode interface. If there is any outside change of what that function would return, the 'dirty' attribute of the WsfGraph instance must be set to '1'.

This object is derived from WsfComponent.

### 46.1.1 var WsfGraph.get_list;

The function which returns the node list. This is set by the constructor to its first argument.

### 46.1.2 var WsfGraph.ibase;

The (relative or absolute) URL to the directory in which the images used by this component can be found.

This are the files 'images/wsf_graph_*' in the SPL sources.

### 46.1.3 var WsfGraph.raster;

Per default, nodes can be moved to any position. It is possible to use a raster to align the nodes. If you want that, set this variable to the size of the raster (e.g. 10 is a good value).

### 46.1.4 var WsfGraph.iframe_header;

Additional HTML text to be inserted in the HTML header of the iframe.

### 46.1.5 var WsfGraph.iframe_footer;

Additional HTML text to be inserted at the end of the HTML text in the iframe.

### 46.1.6 var WsfGraph.iframe_attributes;

Additional attributes to be added in the <iframe> tag. Usually this is something like 'style="width:100%; height:100%"'.

### 46.1.7 var WsfGraph.scroll_x, scroll_y;

Some changes require a reload of the iframe. The current scroll position is then preserved using this attributes. It is also possible to adjust this values when setting the 'dirty' variable to '1'.

### 46.1.8 method WsfGraph.iframe_main();

The main method for the iframe.

This is called whenever the iframe content is reloaded and prints the new content of the iframe.

### 46.1.9 method WsfGraph.get_html();

Overloaded WsfComponent.get_html().

### 46.1.10 method WsfGraph.main();

Overloaded WsfComponent.main().

### 46.1.11 method WsfGraph.destroy();

Overloaded WsfComponent.destroy().

### 46.1.12 method WsfGraph.init(_get_list, _ibase);

The Constructor.

The parameters are copied to the get_list and ibase variables.

The first parameter must be a function pointer. This function, when called, must return an array of objects implementing the WsfGraphNode interface. This objects describe the entire graph and also provide callbacks to react on user events.

## 46.2 interface WsfGraphNode

The node objects used by WsfGraph must implement this interface. Some of the methods in this interface are optional.

### 46.2.1 method WsfGraphNode.get_xywh();

This method returns the position and size of the node in pixel. The return value must be a hash with the keys 'x' and 'y' for the x- and y-coordinate of the left upper corner (0,0 is the left upper corner of the drawing area) and 'w' and 'h' for the width and height.

### 46.2.2 method WsfGraphNode.set_xy(x, y);

When a node is moved or clicked, this callback is called. The new x- and y-coordinates are passed as parameters.

### 46.2.3 method WsfGraphNode.get_links();

This method returns an array with the IDs of the nodes this node has connections to. The IDs must correspond with the keys in the array returned by WsfGraph.get_list.

### 46.2.4 method WsfGraphNode.set_link(trg);

This callback is called when one of the connections of the node are clicked. The ID (see get_links) of the target node of the connection is passed as parameter.

This method is optional. WsfGraph will not try to call it when is is not declared.

### 46.2.5 method WsfGraphNode.get_color();

This method must return the background color for this node. The return value is a string in the usual HTML color '#RRGGBB' format.

This method is optional. WsfGraph will not try to call it when it is not declared.

### 46.2.6 method WsfGraphNode.get_html();

This method must return the HTML content for this node. Usually this is just a label text, but it can any HTML code.

This method is optional. WsfGraph will not try to call it when it is not declared.

# Chapter 47

# Module "wsf_menu"

A module which implements a WSF Component for DHTML menus.

## 47.1 object WsfMenu WsfComponent

This WSF Component adds a DHTML menu to the application window. It should be the first child of the root component and the root component shouldn't add any content before the html code created by this component. Otherwise it is possible that the dynamic module is rendered over that content and covers it.

Derived from WsfComponent

### 47.1.1 var WsfMenu.nojsmenu = 0;

Create boring non-javascript menus if this is set to '1'. Passing 'nojsmenu: 1' to the constructor sets this this variable.

### 47.1.2 var WsfMenu.size_x = 200;

The width of a menu item.

Only adjust this if you have extraordinary long descriptions of your menu items.

### 47.1.3 var WsfMenu.size_y = 25;

The height of a menu element.

Usuall you do not need to change this to another value;

### 47.1.4 method WsfMenu.main();

The overloaded WsfComponent.main().

### 47.1.5 method WsfMenu.get_html();

The overloaded WsfComponent.get_html().

### 47.1.6 method WsfMenu.init(menu_data, %options);

The constructor.

The parameter is a pointer to a hash with the menu structure. The hash key is the description HTML text of the entry (encode it with encode_xml() if needed) and the value is a function pointer for regular menu entries and hashes again for submenus. E.g.:

```
function get_menu_action(text) {
      return function() {
            debug text;
      };
}

var menu_data = [
      'Menu 1' => [
            'Entry #1' => function() {
                  debug "Hello World!";
            },
            'Entry #2' => function() {
                  debug "This is a small demo!";
            }
      ],
      'Menu 2' => [
            'Closure Example #1' =>
                  get_menu_action("Hello World!"),
            'Closure Example #2' =>
                  get_menu_action("Another Test!")
      ],
      'About' => function() {
            debug "This is a demo program!";
      }
];

var menu = new WsfMenu(menu_data);
```

SPL has support for closures. In this example, the get_menu_action() function is using the concept of closures for storing additional data with the function pointers assigned to the items in 'Menu 2'.

Right now, 'nojsmenu: 1' is the only supported option. It sets the nojsmenu variable.

# Chapter 48

# Module "wsf_switch"

This module implements a WSF Component for switching between other components. Usually this is used for navigations switching between differnet dialogs.

## 48.1 object WsfSwitch WsfComponent

This WSF Component provides an easy infrastructure for switching between other components. In most cases, one will derive another object from this one and use that then.

The children["content"] (see WsfComponent.children) contains the current content of this component. Additional children may be defined and are not handled specially by this component.

An example:

```
object Foobar WsfSwitch
{
      var cases = [
            [      'title' => 'Goto Demo A',
                  'code' => function()
                        { return new WsfDisplay("Demo A"); }
            ],
            [      'title' => 'Goto Demo B',
                  'code' => function()
                        { return new WsfDisplay("Demo B"); }
            ]
      ];

      method get_html() {
            switch_reset();
            return
            <>
                  <div id="$id">
                  <spl:foreach var="i" list="cases">
                        <a ${switch_href(i)}>${cases[i].title}</a>
```

```
                    </spl:foreach>
                    $( if (declared children["content"])
                          return children["content"].get_html_cached(); )
                    </div>
                </>;
        }
    }
```

This Object is derived from WsfComponent.

### 48.1.1 var WsfSwitch.cases;

This variable is a hash or array of the possible choices.

Each element is a hash and has an element with the key 'code' which is a function pointer. This function must return the new component which should replace the current children["content"].

Additional keys might be defined in addition to the 'code' key. Usually keys such as 'title' or 'admin_only' are added.

This object creates the key 'isactive' to store information about a choice beeing given the user and later only allows the user entering those choices.

### 48.1.2 method WsfSwitch.main() ;

Overloaded WsfComponent.main().

### 48.1.3 method WsfSwitch.switch_code(code);

This method is executed when one of the choices are selected. But it is also possible to call this method from outside and so forcing a switch to another component (possibly not in the choices given by the elements of the cases variable).

### 48.1.4 method WsfSwitch.switch_href(i) ;

Used to add the href attriuted in a link to one of the choices. See the generic description of this object WsfSwitch for a usage example.

### 48.1.5 method WsfSwitch.switch_reset() ;

Reset the 'isactive' flags (see cases).

This should be called by the get_html() method in derived objects. (see WsfComponent.get_html())

# Chapter 49

# Module "xml"

A module for accessing XML data

This module currently implements XML, XPath, XSLT (including some EXSLT extensions), XML Namespaces and a simple interface for modifying XML data.

## 49.1 builtin xml_parse(xml_text);

This function parses the XML text passed as parameter and returns an XML document handler. This document handler can then be used to access the data in the XML file using XPATH queries. The XPATH queries are used like array indexes:

```
var xmldoc = xml_parse(file_read("demo.xml"));

foreach[] i (xmldoc["//@*"].nodes)
      debug "Attribute '$i' has the value '${xmldoc[i].data}'.";
```

Virtual member variables (.nodes and .data in the example above) are used to specify what kind of data should be accessed. The following virtual member variables are defined for read accesses:

```
.nodes
      Returns an array with references to all nodes in the XML
      tree which do match the XPATH query. This references are
      auto-generated unique identifiers for the nodes which are
      accepted as array indexes just like xpath queries.
      (The 'i' variable in the example above.)

.node
      Only the first matching node.

.type
      The type of the XML node (e.g. "element", "attribute",
```

```
      "text", "cdata", etc).

.name
      The name of the XML node (for element and attribute nodes).

.path
      An auto-generated xpath query to this XML node.

.data
      The data associated with the first matching XML node. This
      is the content if the match is an element node or text node
      and a the value if the match is an attribute node.

.xml
      The XML text of the first matching XML node and its child
      nodes.

.innerxml
      The XML text of the child nodes of the first matching XML
      node.
```

This member variables (except the .nodes variable) do only use the first match. It is possible to specify a seperator using an additional virtual index. Then all matches are included in the result set, seperated by the specified seperator:

```
debug "All attributes: ${xmldoc["//@*"].data.[", "]}";
```

The following virtual memeber variables are supported for writing:

```
.data
.xml
.innerxml
      This variables are also available for writing and have
      the same meaning as for read accesses.

.add_xml_before
.add_xml_after
      This adds new XML text before or after the found matches.

.add_xml_top
.add_xml_bottom
      This adds new XML text as new child nodes before of after
      the child nodes of the found matches.
```

All write queries modify each found match. In cases such as the following example you need to use an iterator because it contains a combined read/write access. Without the iterator, the regex would read the first match, do the substitution once and write the result to all matches:

```
foreach[] i (xmldoc["//@year"].nodes)
      xmldoc[i] =~ s/2004/2005/;
```

The default behavior - if no virtual member varable is specified - is the behavior of the .data variable.

It is also possible to remove XML nodes using the 'delete' SPL keyword:

```
delete xmldoc["//record[@year < 2005]"];
```

The 'node handlers' returned by the .nodes and .node variables can also be used directly to make relative xpath queries. So the following two code snippets are identical:

```
foreach[] n (xmldoc["//data"].nodes)
      debug xmldoc["$n/@value"];

foreach[] n (xmldoc["//data"].nodes)
      debug n["@value"];
```

Check out the documents on www.w3.org for a detailed description of XML and the XPath query language.

## 49.2 builtin xml_dump(xmldoc);

This function expects an XML document handler (as returned by xml_parse) as parameter and creates XML text from it.

The xml is dumped with indenting spaces if the named parameter 'format' is passed and has a 'true' (non-zero) value.

## 49.3 builtin xml_xslt_text(xmldoc, stylesheet, %params);

This function performs an XSLT transformation:

```
var xmldoc = xml_parse(file_read("demo.xml"));
var stylesheet = xml_parse(file_read("demo.xsl"));

debug xml_xslt_text(xmldoc, stylesheet, foo: "'bar'");
```

The first parameter is the handler of XML document to be transformed, as returned by xml_parse(). The 2nd parameter is the XML document handler for the XSLT stylesheet.

Parameters for the stylesheet are passed as named function parameters. Note that the parameters are XPath expressions. I.e. a string constant needs to be passed with quotes.

The return value is the XML text after the transformation.

## 49.4 builtin xml_xslt_xml(xmldoc, stylesheet, %params);

This function works exactly like xml_xslt_text, but it returns a new XML document handler.

## 49.5 object XmlEx

An instance of this object is thrown on XML parser errors.

### 49.5.1 var XmlEx.description;

A description text describing the error.

# Chapter 50

# SPL C-API Documentation

SPL has a pretty simple C interface which allows easy embedding of the SPL virtual machine and the SPL compiler in other applications and extending SPL using functions and modules written in C.

A very complete example program (using almost the entire API) is "splrun.c". It is the tool which is usually used to run SPL programms on the command line.

A rather short example for executing SPL code is "examples/c-api-test4.c". Beware of the other C API demo programms in "examples/": They are test cases for fiddling with SPL byte code and SPL assembler code. Usually one does not want to use SPL on this level.

A nice example for writing SPL modules in C is "spl_modules/mod_termio.c".

All SPL data structures and public functions are defined in the "spl.h" header file.

## 50.1 Running SPL Scripts

Running an SPL Script is easy: The SPL compiler compiles the SPL Script to SPL Assembler commands. But this assembler commands are not genereted as text ouput, they are passed to an SPL Assembler instance. Optionally it is possible to run the SPL optimizer over the generated assembler code:

```
char *spl_source = "debug 'Hello World!';";

struct spl_asm *as = spl_asm_create();

if (spl_compiler(as, spl_source, "spl_script", spl_malloc_file, 1)) error();
spl_asm_add(as, SPL_OP_HALT, 0);
spl_optimizer(as);
```

The first two argument to spl_compiler() are the SPL Assembler instance and the SPL Source to be compiled. The third parameter is the name of the script file (this is primarly used for error messages). The fourth parameter is a function which can be used by the compiler for loading external files. Loading external files is not allowed if this is a NULL pointer. The last parameter is a bool value which specifies if the compiler should generate debug symbols.

The SPL Assembler instance can then be used to dump SPL bytecode. This bytecode is then passed to a newly generated task of an SPL Virtual Machine:

```
struct spl_vm *vm = spl_vm_create();
struct spl_task *task = spl_task_create(vm, "main");

spl_task_setcode(task, spl_asm_dump(as));
task->code->id = strdup("spl_script");
spl_asm_destroy(as);
```

Before the script is executed it is required to configure the SPL Virtual Machine. Usually one wants to activate the standard builtin functions and set a search path for loading additional SPL Modules:

```
spl_builtin_register_all(vm);
asprintf(&vm->path, ".:./spl_modules:%s", spl_system_modules_dir());
```

When it should be possible to execute SPL callbacks from C funktions then one also needs to define a runloop function which should be used for that purpose. Usually the pre-defined 'spl_simple_runloop' function is used for that:

```
vm->runloop = spl_simple_runloop;
```

Now it is possible to execute the script by calling spl_exec() until the HALT opcode is reached. This opcode removes the code page from the current task because there is nothing left to be executed on the page:

```
while ( task->code ) {
      spl_gc_maybe(vm);
      task = spl_schedule(task);
      if ( spl_exec(task) < 0 ) break;
}
```

The spl_gc_maybe() function runs the garbage collector when it is time to do so. The spl_schedule() function does the scheduling between the tasks. For a single-threaded script like in this example it is a NO-OP.

Finally cleaning up all the SPL data structures is easy. Simply destroying the SPL Virtual Machine also destroys everything which is connected to it (including stuff such as unloading modules loaded by the script):

```
spl_vm_destroy(vm);
```

Have a look at "splrun.c" in the SPL source tree for a very complete example of an SPL runtime implementation.

An application which is embedding SPL should be compiled an linked with the options printed by "spl-config −−cflags", "spl-config −−ldflags" and "spl-config −−ldlibs".

## 50.2 Writing SPL functions in C

We will start with an example. In this example we create two new SPL functions named 'myadd' and 'mysub' which do implement integer addition and substraction. First we need to create a C function which implements myadd() and mysub():

```
struct spl_node *spl_builtin_myaddsub(struct spl_task *task, void *data)
{
      int a = spl_clib_get_int(task);
      int b = spl_clib_get_int(task);

      if (!strcmp(data, "add"))
            return SPL_NEW_INT(a + b);

      if (!strcmp(data, "sub"))
            return SPL_NEW_INT(a - b);

      return 0;
}
```

We also could have written two small C functions for the two SPL functions, but then we would have had no example for the second parameter to SPL C functions.

All SPL C functions have the same prototype: They return an SPL node pointer and expect an SPL task struct as first and a void pointer as second argument.

An SPL node pointer is the abstract representation of a "value" in SPL. The task struct contains all information about the currently running SPL task.

Functions such as 'spl_clib_get_int(task)' can be used to pop the arguments from the VM stack. Arguments are pushed from right to left on the machine stack so the first argument must be popped first, then the second, and so on. Scalar values can be popped using the tree functions:

```
int spl_clib_get_int(struct spl_task *task);
double spl_clib_get_float(struct spl_task *task);
char *spl_clib_get_string(struct spl_task *task);
```

You don't need (better said: must not) free the pointer returned by spl_clib_get_string() it is automatically freed as soon as control is passed back to the SPL virtual machine. You also must not modify the string.

In addition to that the three functions

```
int spl_clib_get_argc(struct spl_task *task);
struct spl_node *spl_clib_get_hargs(struct spl_task *task);
struct spl_node *spl_clib_get_node(struct spl_task *task);
```

can be used to get the number of remaining arguments on the VM stack, get the hash with the named arguments or pop an argument as SPL node from the stack respectively. The spl_clib_get_node() function is only needed when complex (non-scalar) data structures are passed as parameters. The return values of

spl_clib_get_hargs() and spl_clib_get_node() must be freed using spl_put() (see the seperate section about SPL data structures below).

New SPL nodes for scalar values can be created using the helper functions

```
struct spl_node *SPL_NEW_INT(int v);
struct spl_node *SPL_NEW_FLOAT(double v);

struct spl_node *SPL_NEW_STRING(char *v);
struct spl_node *SPL_NEW_STRING_DUP(const char *v);
struct spl_node *SPL_NEW_SPL_STRING(struct spl_string *v);
```

The SPL_NEW_STRING() function expects the parameter to be already malloced. The SPL_NEW_STRING_DUP() function is internally using strdup() to create a seperate malloced copy of the string. SPL is internally using binary trees with reference counters for representing strings. That improves the performance of string concatenations massively. If you have created already an SPL string you can easily create a value for it using the SPL_NEW_SPL_STRING() function.

It is also possible to return a NULL pointer in a SPL C function. This is automatically converted to an 'undef' value in SPL.

Finally the new functions need to be registered with the virtual machine. This is done using the spl_clib_reg() function:

```
spl_clib_reg(vm, "myadd", spl_builtin_myaddsub, "add");
spl_clib_reg(vm, "mysub", spl_builtin_myaddsub, "sub");
```

The first parameter is the SPL VM, the second the name of the function and the third a pointer to the C function implementing the SPL function. The last parameter (a void pointer) is simply passed as second parameter to the C function implementing the SPL function whenever it is called. This way our spl_builtin_myaddsub() function can distinguish if it has been called as myadd() or mysub().

A good place for the spl_clib_reg() calls is right after calling spl_builtin_register_all() for the virtual machine.

## 50.3 Writing SPL Modules in C

Extending SPL as described above is only possible when one is embedding SPL in one owns project. But usually one is using an already existing SPL runtime environment an simply wants to add a few functions. This can be done by writing SPL modules. Here is the sourcecode of a simple example module, mod_hello.c:

```
#include <spl.h>
#include <stdio.h>

static struct spl_node *handler_hello(struct spl_task *task, void *data)
{
      printf("Hello World!\n");
      return 0;
```

```
}

void SPL_ABI(spl_mod_hello_init)(struct spl_vm *vm,
             struct spl_module *mod, int restore)
{
      spl_clib_reg(vm, "hello", handler_hello, 0);
}

void SPL_ABI(spl_mod_hello_done)(struct spl_vm *vm, struct spl_module *mod)
{
      return;
}
```

The SPL_ABI() is a macro which add a little prefix with the SPL ABI version to the identifier passed as argument. This ensures that a SPL runtime never loads a module which has been compiled for another SPL ABI. Every module must export the functions 'spl_mod_<module-name>_init' and 'spl_mod_<module-name>_done'.

The _init function is called when a virtual machine loads the module and the _done function is called when a virtual machine which has loaded the module is going to be destroyed.

The third argument to the _init function is set to 1 when the module is loaded while restoring a dumped session. When a module is e.g. creating SPL variables when loaded they get dumped and restored automatically and so do not need to be recreated when the module is loaded while restoring the session.

Compiling and loading the module is straight forward:

```
$ gcc -shared mod_hello.c -o mod_hello.so

$ splrun -q 'load "hello"; hello();'
Hello World!
```

Note that the module *.so file must be named mod_<module-name>.so and that the <module-name> in the filename and the _init and _done functions must be identical.

## 50.4 SPL Data Structures

All SPL values are stored in SPL nodes (struct spl_node). Basically the following data structure is associated with an spl_node:

```
- scalar values (string, integer and floating point)
- a hash with ordering information with references to other nodes
- a context and a class pointer (also references to other nodes)
- the context type information (function context, object, etc.)
- a code pointer (reference to a code page and an address in it)
- a flags field (additional type information and various other stuff)
- data for hosted nodes (see spereate section below)
- some internal data (e.g. for garbage collection)
```

Most of this information is not accessed directly but using helper functions. We have seen already the helper functions for creating new SPL nodes with scalar values. The helper functions for reading the scalar value from an existing node are:

```
int spl_get_int(struct spl_node *node);
double spl_get_float(struct spl_node *node);
char *spl_get_string(struct spl_node *node);

struct spl_string *spl_get_spl_string(struct spl_node *node);
char *spl_get_value(struct spl_node *node);
int spl_get_type(struct spl_node *node);
```

The first three functions are self explainatory. While spl_get_string() returns a single continous string (you must not free or modify this string), spl_get_spl_string() returns the spl_string structure used internally in SPL to store strings. The spl_get_value() function does the same as the spl_get_string() function but returns a NULL pointer when the value is undefined (spl_get_string() return "" in this case). The spl_get_type() is used by the dynamically typed operators to decide which typed operator should be used and returns one of SPL_TYPE_NONE, SPL_TYPE_INT, SPL_TYPE_FLOAT or SPL_TYPE_OBJ. Usually SPL_TYPE_NONE is interpreted as if SPL_TYPE_INT were returned.

SPL is using a hybrid garbage collector which also performs reference counting. Thus one needs to update the reference counters whenever creating new or removing old references for an SPL node. This can be done using the following two functions:

```
struct spl_node *spl_get(struct spl_node *node);
void spl_put(struct spl_vm *vm, struct spl_node *node);
```

The spl_get() function creates a new node when called with a NULL pointer as argument. The argument 'vm' to spl_put() is required for the garbage collector. For this and simmilar reasons most functions for working with SPL nodes require either a pointer to the SPL virtual machine or a pointer to the currently running SPL task as argument. SPL nodes generated with one of the SPL_NEW_*() functions described above have their reference counter already set to one. So it is an error to make an additional call to spl_get() for this nodes unless you are creating more than one link to the new node.

Each SPL node may have child nodes. Such child nodes can be created, looked up and removed using the following three functions:

```
struct spl_node *spl_create(struct spl_task *task,
            struct spl_node *node, const char *key,
            struct spl_node *newnode, int flags);

struct spl_node *spl_lookup(struct spl_task *task,
            struct spl_node *node, const char *key, int flags);

void spl_delete(struct spl_task *task,
            struct spl_node *node, const char *key);
```

The 'flags' argument to spl_create() and spl_lookup() are bitmap. The following values are supported by both functions:

```
SPL_LOOKUP_TEST
      Do not trigger a runtime error when the entry looked for can
      not be found. This is example given used by the SPL 'declared'
      statement.

SPL_LOOKUP_NOCTX
SPL_LOOKUP_NOSTATIC
      These are used internally in recursive spl_lookup() calls,
      example given when looking something up in the class
      derivation path.
```

In addition to the SPL_LOOKUP_TEST flag the following flags are supported by the spl_create() function:

```
SPL_CREATE_LOCAL
      Create it directly here. Do not follow the context pointers
      and skip the local blocks. Do not trigger a runtime error
      when the variable does not exist yet. In most cases one needs
      to pass this flag.

SPL_CREATE_BEGIN
      When creating a new key, create it at the begin of the key
      list. Without this flag new keys are added to the end of the
      key list.

SPL_CREATE_NOSTATIC
      Used internally.

SPL_CREATE_FUNCLOCAL
      Create function local unless already defined in a local
      command block. This is used internally for storing regular
      expression results.
```

The following two functions from the "time" module are used to convert a UNIX 'tm' struct to an SPL data structure and vice versa. I think this is an excelent example for using spl_lookup() and spl_create():

```
static void convert_node_to_tm(struct spl_task *task, struct spl_node *node, struct
{
    memset(ttm, 0, sizeof(struct tm));
    ttm->tm_sec   = spl_get_int(spl_lookup(task, node, "sec",   SPL_LOOKUP_TEST));
    ttm->tm_min   = spl_get_int(spl_lookup(task, node, "min",   SPL_LOOKUP_TEST));
    ttm->tm_hour  = spl_get_int(spl_lookup(task, node, "hour",  SPL_LOOKUP_TEST));
    ttm->tm_mday  = spl_get_int(spl_lookup(task, node, "mday",  SPL_LOOKUP_TEST));
    ttm->tm_mon   = spl_get_int(spl_lookup(task, node, "mon",   SPL_LOOKUP_TEST));
    ttm->tm_year  = spl_get_int(spl_lookup(task, node, "year",  SPL_LOOKUP_TEST));
    ttm->tm_wday  = spl_get_int(spl_lookup(task, node, "wday",  SPL_LOOKUP_TEST));
    ttm->tm_yday  = spl_get_int(spl_lookup(task, node, "yday",  SPL_LOOKUP_TEST));
```

```
    ttm->tm_isdst = spl_get_int(spl_lookup(task, node, "isdst", SPL_LOOKUP_TEST));
    spl_put(task->vm, node);
}

static struct spl_node *convert_tm_to_node(struct spl_task *task, struct tm *ttm)
{
    struct spl_node *result = spl_get(0);
    spl_create(task, result, "sec",   SPL_NEW_INT(ttm->tm_sec),   SPL_CREATE_LOCAL);
    spl_create(task, result, "min",   SPL_NEW_INT(ttm->tm_min),   SPL_CREATE_LOCAL);
    spl_create(task, result, "hour",  SPL_NEW_INT(ttm->tm_hour),  SPL_CREATE_LOCAL);
    spl_create(task, result, "mday",  SPL_NEW_INT(ttm->tm_mday),  SPL_CREATE_LOCAL);
    spl_create(task, result, "mon",   SPL_NEW_INT(ttm->tm_mon),   SPL_CREATE_LOCAL);
    spl_create(task, result, "year",  SPL_NEW_INT(ttm->tm_year),  SPL_CREATE_LOCAL);
    spl_create(task, result, "wday",  SPL_NEW_INT(ttm->tm_wday),  SPL_CREATE_LOCAL);
    spl_create(task, result, "yday",  SPL_NEW_INT(ttm->tm_yday),  SPL_CREATE_LOCAL);
    spl_create(task, result, "isdst", SPL_NEW_INT(ttm->tm_isdst), SPL_CREATE_LOCAL);
    return result;
}
```

The 'keys' used in spl_create() and spl_lookup() are simple strings in which the dot character can be used as seperator for recursive lookups. Example given the following spl_lookup() call looks up the variable 'A', then inside of 'A' the variable 'B' and finally inside of that 'B' the variable 'C':

```
spl_lookup(task, node, "A.B.C", 0);
```

In addition to that there are some special keys starting with the '!' prefix (example given '!CLS' looks up the parent class of a class or object) or the '#' prefix (special variables generated by the compiler). The following two functions can be used to encode everything which is not a number, 7-bit ascii letter or underscore and decode such an encoded again:

```
char *spl_hash_encode(const char *source);
char *spl_hash_decode(const char *source);
```

The algorithm of spl_hash_encode() is also used when hash elements are addressed using the square brakets in SPL. The return value of this functions is a malloced string which must be freed by the caller.

## 50.5 SPL Strings

SPL stores strings in spl_string structs. Such a struct has a reference counter and (optionally) a left child, a right child and a text value. The string represented by such an spl_string struct is the concatenation of the string represented by the left child, the text value and the string by the right child. The following function can be used to create a new string struct:

```
struct spl_string *spl_string_new(int flags,
            struct spl_string *left, struct spl_string *right,
            char *text, struct spl_code *code);
```

The 'flags' argument is a bitmap for which the following values are supported:

```
SPL_STRING_STATIC
      Usually the 'text' argument is a malloced pointer which is
      automatically freed when the spl_string struct is destroyed.
      This flag must be set when the SPL string subsystem should
      not free the string.

SPL_STRING_DOTCAT
      Automatically insert a dot character between the left child
      and the text value. This is used to speed up the creation
      of variable lookup paths (see spl_lookup() above).

SPL_STRING_UTF8
      Indicates that the string contains UTF8 characters. This flag
      is automatically maintained by the strings library and does not
      need to be passed to spl_string_new().

SPL_STRING_NOAUTOGET
      Do not increment the reference counters of the children.
```

The 'left' and 'right' arguments do specify the left and right children and must be NULL when no left or right childer exists.

The 'code' argument is used when the 'text' pointer points to the data section of a code page and the reference counter of the code page should be decremented when this spl_string struct gets destoyed. This argument is usually simply set to NULL.

Often the 'text' argument is dynamically created. There is a printf-like helper function for automatically allocating the text argument and filling it with the string sprintf() would generate:

```
struct spl_string *spl_string_printf(int flags,
            struct spl_string *left, struct spl_string *right,
            const char *fmt, ...);
```

Sometime one needs to manually increment or decrement the reference counter of an SPL string. This can be done using the following two functions:

```
struct spl_string *spl_string_get(struct spl_string *s);
void spl_string_put(struct spl_string *s);
```

Finally there is a function for generating the flat string representation of an SPL string:

```
char *spl_string(struct spl_string *s);
```

This function stores the flat representation of the string as the new text value and dereferences the children. That also means that the caller must not modify or free the return value of spl_string().

## 50.6 Hosted SPL Nodes

Lets have a look at the following SPL/SDL (SDL is the "Simple DirectMedia Library") example program:

```
load "sdl";

sdl_init(640, 480, fullscreen: 0);
sdl_title("Sprites Demo");

var sprite_background = sdl_sprite_create();
var sprite_ball       = sdl_sprite_create();

sprite_background.image = sdl_image_load("background.png");
sprite_ball.image       = sdl_image_load("ball.png");

while (1)
{
    sprite_ball.x = (sprite_ball.x + 5) % 640;
    sprite_ball.y = (sprite_ball.y + 1) % 480;

    sdl_sprite_update();
    sdl_delay(25);
}
```

You can easily see that the SPL nodes pointed to by the variable names 'sprite_background' and 'sprite_ball' are somewhat special. Changing the values of the member variables 'x' and 'y' have the side effect of changing the position of the sprite on the screen. One playing with it a bit will also find some other unusal effects. Example given it is not possible to create new member variables of this structures and the "x" and "y" members seam to only accept integer values.

The sdl_sprite_create() function creates a so-called "Hosted Node", an SPL node that is not a freestanding general purpose data structure but has a close connection to a set of SPL functions or an SPL module. Here is the C sourcecode of the sdl_sprite_create() function (with the parts which are specific to the SDL module and not interesting here removed):

```
static struct spl_node *handler_sdl_sprite_create(struct spl_task *task, void *data)
{
    struct sdl_sprite_hnode_data *hnd =
            calloc(1, sizeof(struct sdl_sprite_hnode_data));

    /* initialize hnd */

    struct spl_node *n = SPL_NEW_STRING_DUP("SDL Sprite");
    n->hnode_name = strdup("sdl_sprite");
    n->hnode_data = hnd;

    return n;
}
```

So a hosted node is an ordinary SPL node with the attribute "hnode_name" set to an identifier specifying the node type, in this case "sdl_sprite". The same naming scheme which applies to global variable and function names does also apply for hnode identifiers.

The spl_node attribute "hnode_data" is a void pinter which can be freely used by the hnode implementation to store local data.

The SDL module also must register a handler function for "sdl_sprite" nodes. This is done using the following function call in the modules _init function:

```
spl_hnode_reg(vm, "sdl_sprite", handler_sdl_sprite_node, 0);
```

The function prototype of handler_sdl_sprite_node() reads:

```
void handler_sdl_sprite_node(struct spl_task *task,
            struct spl_vm *vm, struct spl_node *node,
            struct spl_hnode_args *args, void *data);
```

The last argument to spl_hnode_reg() is simply passed thru to the handler_sdl_sprite_node() function (the 'data' argument). The hanlder function is called whenever an action is performed on an "sdl_sprite" hnode. The action itself is passed using a spl_hnode_args struct and return values are also passed back using that struct:

```
struct spl_hnode_args {
      int action;
      const char *key;
      int flags;
      struct spl_node *value;
};
```

The action field specifies the kind of action that should be performed. The meaning of the other fields depends on the action. Following actions are defined:

```
SPL_HNODE_ACTION_LOOKUP
      Lookup a member variable of the hnode. The "key" field contains
      the name of the member variable and the "flags" field the value
      of the flags argument to spl_lookup(). The value of the member
      should be passed back using the "value" field.

SPL_HNODE_ACTION_CREATE
      Like SPL_HNODE_ACTION_LOOKUP but the "value" field contains the
      value to be set for the member.

SPL_HNODE_ACTION_DELETE
      Deletion of a member has been requested. The "key" field
      contains the name of the member that should be deleted.

SPL_HNODE_ACTION_NEXT
SPL_HNODE_ACTION_PREV
      For listing the member variables. The "key" field contains
      the name of a member. The "value" field should be set to a
```

```
        spl node as returned by SPL_NEW_STRING() which contains the
        name of the next or preview member respectively.

        When the "key" field is a null pointer the first member should
        be returned for SPL_HNODE_ACTION_NEXT and the last member for
        SPL_HNODE_ACTION_PREV.

    SPL_HNODE_ACTION_COPY
        Someone tries to copy the hosted node using the ':=' operator.
        Create a full copy of this node and return the new hosted node
        in the "value" field.

    SPL_HNODE_ACTION_PUT
        This node is going to be destroyed. Free all resources
        which are connected to this node.

    SPL_HNODE_ACTION_DUMP
        The virtual machine state is going to be dumped. Serialize
        the state of this node and set the spl_node attribute
        'hnode_dump' to a string representing the serialized node.

        Be prepared that this handler can later be called with
        'hnode_data' beeing a NULL pointer. Then the node status must
        be restored from the string in 'hnode_dump'.
```

Hosted nodes can be created so they look pretty simmilar to normal SPL data structures (as the SDL sprites described above) or very different (e.g. XML document handlers can be used like hashes with XPath queries as key values).

Most hosted nodes do not implement all of the actions described above. E.g. the NEXT and PREV actions are rarely implemented. Sometimes it is to difficult or even impossible to implement dump and restore. In this cases it is also possible to not implement SPL_HNODE_ACTION_DUMP which then triggers a runtime error when one tries to dump a virtual machine with active hosted nodes without a valid 'hnode_dump' attribute.

## 50.7 Error Handling

There are two ways of error reporting in SPL: Creating VM runtime errors, warnings or debug output and throwing exceptions. In SPL scripts the runtime messages can be produced using the 'panic', 'warning' and 'debug' statements and exceptions can be thrown using the 'throw' statement.

VM runtime errors, warings and debug output can be created using the spl_report() function. Example given:

```
spl_report(SPL_REPORT_RUNTIME, task, "This is an error!\n");

spl_report(SPL_REPORT_RUNTIME|SPL_REPORT_WARNING, task,
            "This is a warning message!\n");

spl_report(SPL_REPORT_RUNTIME|SPL_REPORT_DEBUG, task,
```

```
                "Did you know that %d + %d is %d?\n", 23, 42, 23+42);
```

The 1st spl_report() call in the example creates a runtime error message. The printed error message will contain a stack backtrace and the virtual machine stops execution.

The 2nd spl_report() call in the example creates a runtime warning message. The printed message will contain a stack backtrace. But the virtual machine will continue execution as if nothing special happened.

The 3rd spl_report() call in the example creates a debug message. The printed message will not contain a stack backtrace and the virtual machine will continue execution.

Note that spl_report() supports the well-known printf format strings.

Exceptions can be thrown using the spl_clib_exception() function. The following small example module (mod_exdemo.c) implements the function exdemo() which can throw an exception.

```
#include <spl.h>

static struct spl_node *handler_exdemo(struct spl_task *task, void *data)
{
    int argc = spl_clib_get_argc(task);

    if (argc != 4)
        spl_clib_exception(task, "ExdemoEx",
            "description",
            SPL_NEW_SPL_STRING(spl_string_printf(0, 0, 0,
                "You passed %d arguments to exdemo().\n"
                "This function must be called with 4 arguments.",
                argc)),
            NULL);

    return 0;
}

void SPL_ABI(spl_mod_exdemo_init)(struct spl_vm *vm,
            struct spl_module *mod, int restore)
{
    if (!restore)
        spl_eval(vm, 0, strdup(mod->name), "object ExdemoEx { }");

    spl_clib_reg(vm, "exdemo", handler_exdemo, 0);
}
```

SPL exceptions are objects and thus we first need to create a class for our example exceptions. This is simply done using spl_eval() in the modules _init function. It is important that this spl_eval() is not executed when a dumped vm machine state is restored. Note that the ExdemoEx class has not properties at all. This is not unusual for exception classes.

The handler_exdemo() function does throw exceptions when it is not called with 4 arguments. The exceptions are thrown using the spl_clib_exception()

helper function. This function is called with the current task pointer and the name of the exception class followed by a variable number of arguments. This additional arguments are pairs of a string pointer holding a property name and an SPL node pointer with the value for this property. The list is terminated with a single NULL pointer. In our example we only set the "description" property. This property is automatically included in the 'uncaught exception' error messages:

```
$ gcc -shared mod_exdemo.c -o mod_exdemo.so

$ splrun -q 'load "exdemo"; exdemo(1, 2, 3);'
SPL Runtime Error: Thrown uncaught exception: [ ExdemoEx ]
>> You passed 3 arguments to exdemo().
>> This function must be called with 4 arguments.
in:         ROOT (byte 28 in code block 'command line')
```

The spl_clib_exception() helper is not 100% equal to the SPL 'throw' instruction. The most important difference is that it does not run the constructor for the new exception object.

## 50.8 Callbacks from C to SPL

Sometimes one needs to pass an SPL callback function to a C function and then call this callback function from the C code. This is tricky, but possible:

```
#include <spl.h>

static struct spl_node *handler_callback(struct spl_task *task, void *data)
{
    struct spl_node *callback = spl_clib_get_node(task);
    struct spl_node *arg1 = spl_clib_get_node(task);
    struct spl_node *arg2 = spl_clib_get_node(task);

    struct spl_task *cb_task = spl_clib_callback_task(task->vm);

    struct spl_asm *as = spl_asm_create();
    spl_asm_add(as, SPL_OP_PUSHC, "retval");
    spl_asm_add(as, SPL_OP_ZERO,  0);
    spl_asm_add(as, SPL_OP_PUSHA, "arg2");
    spl_asm_add(as, SPL_OP_PUSHA, "arg1");
    spl_asm_add(as, SPL_OP_DCALL, "callback");
    spl_asm_add(as, SPL_OP_POPL,  0);
    spl_asm_add(as, SPL_OP_HALT,  0);
    spl_task_setcode(cb_task, spl_asm_dump(as));
    spl_asm_destroy(as);

    spl_create(cb_task, cb_task->ctx, "callback", callback, SPL_CREATE_LOCAL);
    spl_create(cb_task, cb_task->ctx, "arg1", arg1, SPL_CREATE_LOCAL);
    spl_create(cb_task, cb_task->ctx, "arg2", arg2, SPL_CREATE_LOCAL);
```

```
    spl_clib_callback_run(cb_task);

    struct spl_node *retval = spl_get(spl_lookup(cb_task,
                                cb_task->ctx, "retval", 0));
    spl_task_destroy(cb_task->vm, cb_task);

    return retval;
}

void SPL_ABI(spl_mod_callback_init)(struct spl_vm *vm,
                                struct spl_module *mod, int restore)
{
    spl_clib_reg(vm, "callback", handler_callback, 0);
}
```

This example module (mod_callback.c) implements a function which should be called with three arguments. The first one is a callback function and the other two are arguments which are passed thru to the callback function. The return value of the callback function is also the return value of the example function. Example given:

```
$ gcc -shared mod_callback.c -o mod_callback.so

$ splrun -q 'load "callback";
             debug callback(function(a,b) { return a+b; }, 23, 42);'
SPL Debug: 65
```

As one can see, it is not possible to call the callback function directly. Instead one needs to create a seperate task (using spl_clib_callback_task()) and generate a little helper codepage which then calls the callback function. This codepage can then be executed using the spl_clib_callback_run() function.

In the example program I have used temporary SPL variables in the local context of the task created by spl_clib_callback_task() to pass the callback, the arguments and the return value to and from the callback codepage. A task context is not very different from any other SPL node. So the variables can be easily created with spl_create() and looked up with spl_lookup().

The SPL assembler code used in the example above is pretty generic and is sufficient for the most cases where an SPL callback should be called from C code. Simply add as many SPL_OP_PUSHA records as you have arguments.

The command 'splrun -AN' can be used to create SPL assembler code from SPL scripts. So that command can be used if the above example does not cover your requirements. Example given the code used in the above example can be generated like this:

```
$ splrun -AN -q 'var retval=callback(arg1, arg2);'
# SPL Assembler Dump
:16     PUSHC       "retval"         # (1 byte arg) 10
:18     ZERO
:19     PUSHA       "arg2"           # (1 byte arg) 0
:21     PUSHA       "arg1"           # (1 byte arg) 5
```

```
:23     DCALL       "callback"       # (1 byte arg) 17
:25     POPL
:26     HALT
# Total size: 37 bytes (11 text, 26 data).
```

One could also call the SPL compiler directly in the C function to generate the SPL assembler code. But the SPL compiler is rather slow and calling it for such a small code snippet is an unnecessary overhead.