

spConfig-0.1.3 API Reference

Christopher Wright, cwright@softpixel.com
Steve Mokris, smokris@softpixel.com

May 23, 2003

Contents

1	Introduction	1
2	Structure	2
2.1	struct spConfigNode	2
2.2	struct spConfigAttribute	3
2.3	struct spConfigMessage	3
3	API	4
3.1	Allocation / Duplication / Deallocation	4
3.2	Node Management	4
3.3	Loading	5
3.4	Saving	5
3.5	Messages	5
3.6	Finding Information	6
3.7	Debugging	6

1 Introduction

To visualize the way configuration files work in spConfig, one simply needs to think of a hierarchical filesystem. Configuration files are arranged in a tree-like structure, composed of ‘nodes’ (analogous to directories), ‘attributes’ (files), and ‘text’ (which is a sort of metadata). Additionally, commands such as ‘notices’ can be inserted into the config tree. There are plans for simple logic commands and an ‘include’ operation in the future.

The syntax is similar to XML, and the parser should be able to grok most XML files. We will perform a more extensive review of this in the future. For a fairly complete example and documentation of the syntax, see demo/demo.conf.

2 Structure

The base of any config tree is a particular node, known as the **root node**. Typical operation involves using the library to read from a config file (on disk or in memory) into a memory structure, then using the library-provided routines to glean information from this structure. It is also possible to use the library-provided routines to construct such a structure in its entirety. In the future, it will be possible to write out these memory structures to disk in our config file format.

2.1 struct spConfigNode

```
typedef struct spConfigNodeS
{
    char          *name;
    char          confFlag;
    char          *text;

    int           numAttributes;
    int           numChildren;
    spConfigAttribute **attributes;
    struct spConfigNodeS **children;

    struct spConfigNodeS *parent;
    spConfigMessage *message;
    void>(*callback)(struct spConfigNodeS*node, char*string);
} spConfigNode;
```

`name` is obviously the name of the node. When reading in a configuration file from disk, the root node's name points to NULL, but this is not a requirement for structures created in memory (more on this below).

`text` stores the text that happened to be located in the node, between the opening tag and closing tag, if any. It does not include text within child nodes.

`numAttributes` number of attributes this node has

`numChildren` number of children this node has

`**attributes` one-dimensional array of attributes

`**children` one-dimensional array of child nodes.

`parent` is a pointer to the parent node of this node, if there is any (NULL if root node). Unfortunately, sharing a node between two parents will result in a pointer to only one.

`message` is the beginning of a linked list of messages attached to this node.

`confFlag` can have any of the following set:

```
SP_CONFIG_NODE_ALLOW_TEXT    - this node accepts text if parsed
SP_CONFIG_NODE_ALLOW_ATTRIBUTES - this node accepts additional attributes
SP_CONFIG_NODE_ALLOW_NODES   - this node accepts additional child nodes
```

These flag values are not yet implemented, however.

`callback` is a function used to handle include files. Because we don't support include files just yet, this field is presently useless.

2.2 struct spConfigAttribute

attributes are name/value pairs defined within the node begin tag.

```
typedef struct
{
    char    *name;
    char    type;
    union
    {
        int i;
        double d;
        char *string;
    } value;
} spConfigAttribute;
```

name stores the attributes name (this should always be set).

type determines which type of data this attribute holds, if any.

```
SP_CONFIG_VALUE_NONE    - no values are associated with this attribute
SP_CONFIG_VALUE_INT     - the value is an int (use value.i)
SP_CONFIG_VALUE_FLOAT   - the value is a double (use value.d)
SP_CONFIG_VALUE_STRING  - the value is a string (use value.string);
```

2.3 struct spConfigMessage

And finally, we have **messages**.

```
typedef struct spConfigMessageS
{
    int    line,col;
    char   type;
    char   *text;
    char   *filename;
    struct spConfigMessageS *next;
} spConfigMessage;
```

line number (of input file/string) upon which this message occurred

col column of message

type determines the severity of the message:

```
SP_CONFIG_MESSAGE_INFO    - information, not a problem
SP_CONFIG_MESSAGE_NOTICE  - small trivial warning, nothing too bad
SP_CONFIG_MESSAGE_WARNING - something may have been misparsed
                           but overall we're still on track.
SP_CONFIG_MESSAGE_ERROR   - something is very wrong, parsing may
                           be wrong after this point.
```

text stores the actual message.

filename stores which file the message originated in (currently useless, we don't support include files yet).

next stores a pointer to the next message, or NULL if there are no more messages.

3 API

3.1 Allocation / Duplication / Deallocation

New

```
spConfigNode      *spConfigNodeNew      ();
spConfigAttribute *spConfigAttributeNew ();
spConfigMessage   *spConfigMessageNew   ();
```

These functions return a newly allocated and initialized item, or NULL on error (out of memory).

Clone

```
spConfigNode      *spConfigNodeClone      (spConfigNode *node);
spConfigAttribute *spConfigAttributeClone (spConfigAttribute *att);
spConfigMessage   *spConfigMessageClone   (spConfigMessage *msg);
```

These functions return a complete copy of the input item. Strings are copied, child nodes and values are copied, and any other pointers are duplicated, not shared. Thus it is safe to clone a tree and hack on one. The other will be preserved.

Copy

```
spConfigNode      *spConfigNodeCopy      (spConfigNode *node);
spConfigAttribute *spConfigAttributeCopy (spConfigAttribute *att);
spConfigMessage   *spConfigMessageCopy   (spConfigMessage *msg);
```

These functions copy the values of the input item, returning a newly allocated item with the same values. Pointers are shared, so copying a tree and hacking on one will hack up the other as well. Very hard to cleanly destruct (pointers could already be freed), these probably shouldn't be used unless you know what you're doing.

Free

```
void spConfigNodeFree      (spConfigNode *node);
void spConfigAttributeFree (spConfigAttribute *att);
void spConfigMessageFree   (spConfigMessage *msg);
```

These functions free the memory used by the item passed to them. Messages and nodes have their children freed as well.

3.2 Node Management

AddChild

```
int spConfigNodeAddChild (spConfigNode *parent, spConfigNode *child);
```

This function adds node **child** to the list of child nodes in **parent**.

Returns 0 on success, else error.

AddAttribute

```
int spConfigNodeAddAttribute (spConfigNode *parent, spConfigAttribute *att);
```

This adds value **att** to the list of attributes in **parent**.

Returns 0 on success, else error.

AddAttribute*

```
int spConfigNodeAddAttributeInt      (spConfigNode*parent, char*name, int i);
int spConfigNodeAddAttributeDouble   (spConfigNode*parent, char*name, double d);
int spConfigNodeAddAttributeString   (spConfigNode*parent, char*name, char*text);
```

These functions add an attribute named **name**, with a value of the item passed, to the list of attributes in **parent**.

They return 0 on success, else error.

NodeName

int **spConfigNodeName** (spConfigNode *node,char *name);

Sets the name of **node** to **name**, freeing the old name if there was one.

Returns 0 on first name, else the old name was freed.

AttributeName

int **spConfigAttributeName** (spConfigAttribute *att,char *name);

Sets the name of **attribute** to **name**, freeing the old name if there was one.

Returns 0 on first name, else the old name was freed.

3.3 Loading

Load

int **spConfigLoad** (spConfigNode *root,char *filename);

This function loads a config file from **filename**, parses it, and stores whatever it successfully parses in **root**.

Returns 0 on success, else error.

LoadStr

int **spConfigLoadStr** (spConfigNode *root,char *name,char *string);

This function is identical to **spConfigLoad** above, but instead of a file it takes a string. this allows the program to store a compressed and/or encrypted configuration file, as long as this function gets plain text for parsing. name is the name that will show up in messages as the file name.

Returns 0 on success, else error.

3.4 Saving

Save

int **spConfigSave** (spConfigNode *root,char *filename);

This function saves a configtree based on **root** in a file named **filename**. This file can then be opened and reparsed with **spConfigLoad()** to return an identical tree at a later time.

Returns 0 on success, else error.

SaveStr (NOT YET IMPLEMENTED)

char ***spConfigSaveStr** (spConfigNode *root);

This function returns a string representing the config file in parsable form. instead of saving the config tree in a file, this allows the program to compress and/or encrypt the config string before saving, effectively protecting the configuration from casual hacking.

3.5 Messages

MessagesList

char ***spConfigMessagesList** (spConfigNode *rootNode);

This function returns a string containing all the messages in **rootNode**. This is suitable for printing on the screen or saving to a file.

MessagesClear

void **spConfigMessagesClear** (spConfigNode*rootNode);

This function removes all the messages from **rootNode** and reclaims the memory used by those messages.

3.6 Finding Information

Find

```
spConfigNode      *spConfigNodeFind  
                  (spConfigNode *root, char *name, char deep, spConfigNode *previous);  
spConfigAttribute *spConfigAttributeFind  
                  (spConfigNode *root, char *name, char deep, spConfigAttribute *previous);
```

These functions search through the entire config tree specified by **root**, looking for nodes or attributes with the name **name**. If **deep** is non-zero, child nodes of **root** are recursively included in the search. If **previous** is non-null, it is assumed to be a pointer to the previous node/attribute returned, and the function will return the next node/attribute after that one. On the first call, **previous** should be NULL.

FindPath

```
spConfigNode      *spConfigNodeFindPath      (spConfigNode *root, char *path);  
spConfigAttribute *spConfigAttributeFindPath (spConfigNode *root, char *path);
```

These functions return a node or attribute pointing to the node or attribute indicated in **path**. **path** is a string similar to a unix path. NULL is returned if nothing is found. If some of the **path** is correct, it returns the closest match (deepest matching directory).

GetPath

```
char *spConfigNodeGetPath (spConfigNode *node);
```

This function returns the path string of **node**.

3.7 Debugging

For detailed information regarding debugging, please see `doc/DEBUGGING`

File Output

```
void DEBUGFD_spcnfig (FILE *debuglog);
```

This is a debugging function used to specify where the debug messages should go. **debuglog** should be a file opened for writing. The default log is `stderr`. **This function has no effect if `spcnfig` is compiled without debugging.**

Debug Level

```
void DEBUGLEVEL_spcnfig (signed char level);
```

This is a debugging function used to specify the verbosity of the debug messages. **level** is a value between -128 and 127. Lower levels result in more messages. The default level is 0. **This function has no effect if `spcnfig` is compiled without debugging.**