

# **SIEGFRIED LOCALE LIBRARY**

## **Developer Guide**

**Copyright by Siegfried Soft**

**Siegfried Soft  
Hauff-Weingärtner GbR  
Reichenberger Str. 12  
D-34246 Vellmar**

## General

### What is the Siegfried Locale Library?

The sfliblocale.so library enable BeOS software developers to include multi- language support to their applications. This is realized by collecting all needed text of an application in an external file. The evaluation of those files (loading data, get text, etc.) will be done by the sfliblocale.so library. By using ID numbers for every text, the library can quickly and easily access the data.

For every language supported a ("locale") file including the translated text is needed. The "locale" files will be created by using the Siegfried Locale Editor.

---

### Terms of use

The sfliblocale.so library, the source code examples and the documentation are copyright by Siegfried Soft. BeOS software developers reserve the right to use the sfliblocale.so library and the source code example in their own projects without obligation to pay any royalty fees. This is valid for private as well as for commercial usage and their give away in compiled form.

---

### Warranty

Siegfried Soft has made every effort to insure the quality and reliability of this product. Siegfried Soft does not accept any liability for damage created by the use or misuse of this product.

---

### Notice / Trademarks

In this manual, we mention several registered trademarks which are not marked as such in the body of the text. Hence you cannot conclude from the missing identification that the corresponding product name is free from rights of third parties:

- Be, BeOS is a trademark of the company Be Inc.

---

### System requirements

The Siegfried Locale Library need BeOS R5 or higher. Siegfried Locale Library supports the x86 platform as well as the PPC.

---

### Error reports, criticism, suggestions

If you should stumble over any error while using the Siegfried Locale Library or if you should have any suggestions for improvement: please write to us! We are open to positive and negative criticism as well as to suggestions. We will try to incorporate your ideas in the Siegfried Locale Library as far as possible. If you should find any error, please specify your computer configuration. Please write to the following address:

**Siegfried Soft**  
**Hauff-Weingärtner GbR**  
**Reichenbergerstraße 12**  
**D-34246 Vellmar (Germany)**

WEB: <http://www.siegfried-soft.de>  
E-Mail: [hauff@siegfried-soft.de](mailto:hauff@siegfried-soft.de)

---

## **Are you aware of Siegfried Backup? If not, take this opportunity to correct this.**

Siegfried Backup is a powerful and easy-to-use application to backup and restore all your software and data. It is the optimal tool for beginners and experts wishing to backup and restore any kind of data speedily and smoothly.



Siegfried Backup contains all the functions demanded of a top quality backup application. Furthermore, special attention has been paid to sophisticated graphic user guidance.

The idea behind the tool is to make work with Siegfried Backup as easy and efficient as possible for beginners as well as for experts. You will appreciate the manifold features offered, long after your initial steps with the tool. A large number of specific functions (such as filters, scheduler, compression, add-ons and many others) will turn your daily work with Siegfried Backup into a genuine pleasure.

Get a demo version of Siegfried Backup now: <http://www.siegfried-soft.de>

**Siegfried Backup, your ticket to modern data management technology!!**

---

## Using the sfliblocale.so library

This chapter contains information about how to include the sfliblocale.so library in projects und how to use the functions of the library.

### The priciple of localization

The kind of localization for the sfliblocale.so library is quiet simple. Every text which should be localized is included in a table. The text is identified by the position in the table (the table row). The main function of the library returns a pointer to the text by using the row number as parameter. A Table must be created for every language that should be supported. Corresponding to the row number the text will be translated to the approriated language. Every table is saved as a single file. To get multi language support only the approriated table ("locale" file) is to be loaded and used. By adding "locale" files an application can be easy expand for new languages, without any additional development effort!

---

### Inserting the library into a project

To use the library in a project, two files are needed:

File	Folder	Comment
sfliblocale.so	develop/lib/x86	Locale Library for x86 plattform
sfliblocale.so	develop/lib/ppc	Locale Library for PPC plattform
SFLocale.h	develop/include	Class/function defintion for the sfliblocale.so

Insert both files simply to the project (by using "Add Files" command of the BeIDE).

---

### Adding the library to an application

The sfliblocale.so library isn't a static link library, it's a dynamic link library. That means that the library is linked to the application during the startup of the program. By this, the library must be accessible. BeOS defines two folders for third party libraries. The first one is a local path "lib" at the folder where the application resides. The second path is "/boot/home/config/lib". That's the folder for shared libraries. If BeOS can't find the library in both folders an error message appears and the application terminates.

Which place for the library is the best can be decided by the developer. We recommended to use the local folder, because of an easier deinstallation. The user has only to delete the application folder to remove all files, he doesn't need to remember where to find other parts of the application.

---

### Preparation/Organisation

For an optimal usage of the library all needed text of an application is to collect in a single (Include) file. A second (include) file should manage the text IDs (symbolic representation of the table row numbers).

The collected text is used as default (built-in) text for the application. If an application can't get an specific text from the "locale" file the built-in default text is used by the library automatically. This can be the case if the application is updated and the "locale" files are not. If an error occurs, the application is usable without "locale" files using the built-in text automatically. Simply the default built-in text is used. To get an application international running it's strongly recommend to use English for the default built-in text.

The naming convention of both files is free. For the example provide by us, we have named the text file "SFTexts.h" and the ID file "SFTextIDs.h".

The files should be have the following structure:

```

#ifndef _SFTEXTS
#define _SFTEXTS

#include "SFTextIDs.h"                                // The text IDs

//-----
static const char    *mText[SF_LAST_ID] =
{
    "Ok",                                //SFOK
    "Cancel",                            //SFCANCEL
    "Continue",                          //SFCONTINUE
    :
    :
    "ERROR: There is no text to save!",    //SFERRNOTEXT
    "Font",                                //SFFONT
    "Size",                                //SFSIZE
};

//-----

#endif

```

If a new text is needed, it is appended to the end of the list and the corresponding ID is inserted to "SFTextIDs.h".

The file "SFTextIDs.h" looks like below:

```

#ifndef _SFTEXTIDS
#define _SFTEXTIDS

#define SF_DEFAULT_LANGUAGE    "English"
#define SF_TEXT_APP_ID        "siegfried localeeditor locale"

#define MSG_SFLANGUAGE        'sflg'

//-----
// Definition for all used text data
//-----

enum gSFTextID
{
    SFOK,
    SFCANCEL,
    SFCONTINUE,
    :
    :
    SFERRNOTEXT,
    SFFONT,
    SFSIZE,

    SF_LAST_ID                // MUST be always the last ID!!!
};

#endif

```

For every new text that is appended to the list of "SFTexts.h" the corresponding symbolic identifier (ID) is insert to "SFTextIDs.h". The ID is to be included at the end of the identifier list (but before label "SF\_LAST\_ID"), because the text position at the list and the position of the identifier must remain the same!

By inserting before the identifier "SF\_LAST\_ID" we automatically get the number of text list entries.

**IMPORTANT:** It's strictly to check that the position of the text and the position of the identifiers are the same. Differences will be result in very strange text outputs of the application :-)

The ID file defines two additional constants:

"SF\_DEFAULT\_LANGUAGE" is the name of the used default build-in language. Normally this should be "English".

The second one is "SF\_TEXT\_APP\_ID". The application ID is a unique identifier for an application that is using "locale" files. The ID shows whether a "locale" file can be used by the application or not. Nothing brings up more strange effects than a "locale" file that isn't directed to the application ;-). The application ID is created by the developer of an application. All "locale" files of an application must have the same ID. Files with other IDs are ignored by the application. The developer is free to decide what kind of string is used for the application ID. It's a good idea if there is a reference to the application. For example, the Siegfried Locale Editor is using as application ID "siegfried localeeditor locale".

It's possible to use as ID the BApplication object signature of the application, because it's suffice unique.

---

## Include the sfliblocale functionality to an application

To use the sfliblocale.so library in applications, at first there is some initialization needed. The best place for this initialization is the startup function of the application, the inherited BApplication class. The initialization need only be done once.

The job of the initialization is to set the built-in text, the name of the build-in language and the application ID for the "locale" files. Thereby that text data and the text IDs are separated into different (Include) files the initialization is simple to realize.

### Example:

```
#include "SuperApp.h"
#include "SFTextIDs.h"
#include "SFTexts.h"
int main()
{
    SuperApp *myApplication;

    myApplication = new SuperApp();           // create/init application
    myApplication->Run();                     // run application

    delete(myApplication);                   // delete application
    return(0);                               // see you!!!
}

SuperApp::SuperApp()
    : BApplication("application/x-vnd.siegfriedsoft-superapp")
{
    SFLocale *lang;
    lang = SFLocale::GetInstance();          // get text data access
    lang->SetAppID(SF_TEXT_APP_ID);          // set application ID
    // --- Set build-in text data ---
    lang->SetDefaultText(mText, SF_LAST_ID, SF_DEFAULT_LANGUAGE);
    // --- Now it's possible to access the text data.
    // --- Initialization finished.
    :
    :
}
```

At this point it's a good idea to load/set the last used language by using the function `"SetLanguage ()"`. After the initialization is finished at every point within the application it's possible to access the text data. Access can be get by the following call:

```
SFLang  *lang;
lang = SFLocale::GetInstance();           // Get text data access
```

It's recommend for classes that need access to the `SFLocale` object to declare a class private or for base classes a protected variable. To initialize the variable use `"GetInstance ()"` at the constructor function of the class (see chapter "Function overview / GetInstance()" for more information).

---

## Usage of the text funtions

Without doubt `"Text ()"` is the most used function of the `sfliblocale.so` library. Every text for output that an application needs is delivered by `"Text ()"`. As a parameter the function needs the ID number of the appropriated text data. The result of the function call is a pointer to a text string. if there was no text data for the used language, the pointer contain the english build-in text.

To localize a `"BButton"` the label must be replaced by the `"Text ()"` function.

### Example:

```
BButton  *button;
button = new BButton(BRect(0, 200, 200, 225), "btn_quit",
                    cLang->Text(SFQUIT), new Message(B_QUIT_REQUESTED),
                    B_FOLLOW_HCENTER+B_FOLLOW_BOTTOM);
```

In the example above the `"SFLocale"` object (`cLang`) is a class private variable and was initialized at the constructor of the class by `"GetInstance ()"`. The text ID `"SFQUIT"` is a user defined constant declared and managed in the file `"SFTextIDs.h"`.

The resulting text pointers of `"Text ()"` can be used in many ways. There is no problem to expand the text:

### Example:

```
BButton  *button;
BString  string;

string = cLang->Text(SFSAVE);
string += "...";
button = new BButton(BRect(300, 10, 400, 30), "btn_filesave",
                    string.String(), new BMessage(MSG_SAVEFILE_PANEL),
                    B_FOLLOW_RIGHT+B_FOLLOW_TOP);
```

This kind of adding has the advantage that the amount of needed text will be reduced. The example above shows only that text `"Save"` is needed and not `"Save"` and `"Save..."`.

Another kind of expanding text is to include numbers. This is simplified through the use of the the C library function `"sprintf ()"`.

### Example:

```
// Text "You have saved %ld files to\n'%s'!"           // SFSAVECOUNT
char  buf[128];
long  count = 6587;
char  *dest = "/boot/home/savefolder";

sprintf(buf, cLang->Text(SFSAVECOUNT), count, dest);
```

**Output:**

You have saved 6587 files to  
'/boot/home/savefolder'!

---



## Changing the language

To change the language the best user guidance is to use a menu. A sub menu contains all available languages with the current one marked. In addition it's a good idea to react to changes in the folder of the "locale" files (e.g. if a new language is added).

### Example:

```
BMenu          *smenu;
const char      *last_language;
:
:
smenu = new BMenu(cLang->Text(SFLANGUAGE)); // create language menu
// get the name of the last used language
last_language = GetAppPreferences(...);
// create items for menu
BuildLanguageMenu(smenu, last_language);
cLocaleMenu = smenu;                      // save pointer to language menu
:
:
```

The function "BuildLanguageMenu()" creates the sub menu. The function walks through the language folder and adds all files with the correct application ID to the menu list. The last used language is also marked. Every menu item corresponds to a BMessage. The "what" field is set to "MSG\_SFLANGUAGE". The message includes an entry that contains the name of the language.

The setting of a new language is done by the inherited "MessageReceive()" function of the application window.

```
void SuperAppWin::MessageReceived(BMessage *msg)
{
    entry_ref    ref;
    app_info     info;
    BPath        path;
    BEntry       entry;
    :
    :
    switch (msg->what)
    {
        :
        :
        //---- Sprache setzen ----
        case MSG_SFLANGUAGE:
            be_app->GetAppInfo(&info);    // get program info
            entry.SetTo(&info.ref);       // get access to program
            entry.GetPath(&path);          // path + filename
            path.GetParent(&path);         // Programmpfad ermitteln
            path.Append("locale");        // get only application path
            cLang->SetLanguage(&path, msg->FindString("language"));
            // save here the name of the language to the preferences
            // SetAppPreferences(...);
            break;
        //---- Default message management ----
        default:
            BWindow::MessageReceived(msg);
    }
}
```

To change a language the function "SetLanguage()" is used. The function needs the path where the "locale" file is found and the name of the new language.

## Managing "locale" files

In general it doesn't care where the "locale" files reside. The library checks if the file is a "locale" file for the application or not. It's a good idea to collect the "locale" files in a single folder. At best it's recommended to use a folder inside of the application folder. If the application resides in `"/boot/home/superapp"`, as path for the "locale" files `"/boot/home/superapp/locale"` can be used. By this all files will be collected in one place enabling the user to easily de-install the application. One only has to remove the application folder, without the need to remember if there are other folders or files to remove. The name "locale" isn't a prescription. But we recommend the use of its name, because it's very intuitive for the user.

The node watching of BeOS is used to watch the folder where the "locale" files reside. The best place to activate the node watching is the constructor function of the application window:

```
SuperAppWin::SuperAppWin(...)  
    : BWindow(...)  
{  
    app_info  info;  
    BEntry    entry;  
    BPath     path;  
    node_ref  nref;  
    :  
    :  
    :  
    // --- activate node watching for the "locale" folde ---  
    be_app->GetAppInfo(&info);           // get program info  
    entry.SetTo(&info.ref);              // get access to program  
    entry.GetPath(&path);                 // path + filename  
    path.GetParent(&path);                // only application path is needed  
    path.Append("locale");               // append language folder  
  
    if (entry.SetTo(path.Path()) == B_OK) // get access to the folder  
    {  
        entry.GetNodeRef(&nref);          // get node_ref for folder  
        // Überwachung aktivieren  
        watch_node(&nref, B_WATCH_DIRECTORY, this);  
    }  
    entry.Unset();  
}
```

By using node watching it's simple to detect changes at the language folder (e.g. files added or removed). If the application detects changes the language sub menu is updated. The interpretation of the node watching message will be done in the `"MessageReceived()"` function of the window class.

```

void SuperAppWin::MessageReceived(BMessage *msg)
{
    BMenuItem *item;
    BString    string;
    int32      opcode;

    switch (msg->what)
    {
        :
        :
        case B_NODE_MONITOR:
            if (msg->FindInt32("opcode", &opcode) == B_OK)
            {
                switch (opcode)
                {
                    //---- file/folder deleted ----
                    case B_ENTRY_REMOVED:
                    //---- file/folder moved ----
                    case B_ENTRY_MOVED:
                    //---- file/folder created ----
                    case B_ENTRY_CREATED:
                        if (cLocaleMenu)
                        {
                            while ((item = cLocaleMenu->RemoveItem((int32)0)) >
                                (BMenuItem *)NULL)
                            {
                                if (item->IsMarked())
                                    string = item->Label();
                                delete item;
                            }
                            BuildLanguageMenu(cLocaleMenu, string.String());
                        }
                        break;
                    }
                }
                break;
            }
            //----
            :
            :
        }
    }
}

```

---

## Scripting for the Siegfried Locale Editor

To get an easy translation of the text data for an application, it's a good idea to copy the built-in text data to the Siegfried Locale Editor as a reference. Normaly the file that contains the built-in text ("SFTexts.h") is loaded to the editor. This is only possible for the developer(s) of the application. All other users can do a translation only if they acquire text data from the developer.

It's possible to simplfy this by using the scripting capabilities of BeOS. Scripting enables every user to directly acquire the built-in text, the language name and the application ID. The Siegfried Locale Editor uses these scripting capabilities to receive the data from every application that supports the sfliblocale.so library correctly.

To make an application locale scripting aware only the inherited "BApplication" class needs to be extended a little bit.

To interpret scripting commands the "MessageReceived()" function must be extended by "B\_GET\_PROPERTY":

```
void SuperApp::MessageReceived(BMessage *msg)
{
    BMessage    spec, reply;
    bool        found;
    int32       index, what;
    const char  *prop;

    switch (msg->what)
    {
        //---- Scripting ----
        case B_GET_PROPERTY:
            found = false;
            if (msg->GetCurrentSpecifier(&index, &spec, &what, &prop) == B_OK)
            {
                reply.what = B_REPLY;
                if (strcmp(prop, "DefaultLanguage") == 0 && what ==
B_DIRECT_SPECIFIER)
                {
                    found = true;
                    reply.AddString("result", SF_DEFAULT_LANGUAGE);
                }
                if (strcmp(prop, "DefaultText") == 0 && what ==
B_DIRECT_SPECIFIER)
                {
                    found = true;
                    for (index = 0; index < SF_LAST_ID; index++)
                        reply.AddString("result", mText[index]);
                }
                if (strcmp(prop, "TextAppID") == 0 && what == B_DIRECT_SPECIFIER)
                {
                    found = true;
                    reply.AddString("result", SF_TEXT_APP_ID);
                }
            }
            if (found)
                msg->SendReply(&reply);
            else
                BApplication::MessageReceived(msg);
            break;
        //---- Default message management ----
        default:
            BApplication::MessageReceived(msg);
    }
}
```

The function will be extended by three scripting commands:

- "DefaultLanguage" returns the Name of the built-in language ("English", "Deutsch").
- "DefaultText" returns the built-in text data as a field.
- "TextAppID" returns the application ID of the "locale" files.

The above used constants "SF\_LAST\_ID", "SF\_DEFAULT\_LANGUAGE" and "SF\_TEXT\_APP\_ID" can be found in the user defined files "SFTexts.h" and "SFTextIDs.h".

In addition, to implement scripting correct the BeOS hook functions "GetSupportedSuites()" and "ResolveSpecifier()" need to be implemented:

```
static property_info mPropList[] = {
    { "DefaultText", {B_GET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "get the
default texts for localization", 0},
    { "DefaultLanguage", {B_GET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "get the
default language name for localization", 0},
    { "TextAppID", {B_GET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "get the id
for locale files", 0},
    0 // terminate list
};

status_t SuperApp::GetSupportedSuites(BMessage *msg)
{
    BPropertyInfo PropInfo(mPropList);

    msg->AddString("suites", "suite/vnd.SiegfriedSoft-locale");
    msg->AddFlat("messages", &PropInfo);

    return BApplication::GetSupportedSuites(msg);
}

BHandler *SuperApp::ResolveSpecifier(BMessage *msg, int32 index, BMessage
*spec, int32 form, const char *prop)
{
    BPropertyInfo PropInfo(mPropList);

    if (PropInfo.FindMatch(msg, index, spec, form, prop) >= 0)
        return this;

    return BApplication::ResolveSpecifier(msg, index, spec, form, prop);
}
```

## Function overview

### SFLocale \*GetInstance()

By using the function "GetInstance()" an application receives access to the text data of the loaded "locale" file. As opposed to other classes the creation of an "SFLocale" object isn't needed (and not possible). Access to the class will be done in every instance by a pointer returned from "GetInstance()". The class "SFLocale()" is a so called *"Singleton design pattern"*. A Singleton design pattern create one (and only one) instance of a class. This behaviour saves memory and if a language change is done the text data remains consistent.

Example:

```
SFLocale *lang;

lang = SFLocale::GetInstance();    // get access to text data
```

Now at every point within the application it's possible to access the text data. Only one instance of the class exists. The pointer to the instance can be a global variable. But that's not the fine art of programming, because the principle of encapsulation is broken. It's better to declare the pointer as class private or for base classes as protected:

```
xyz {
    xyz();
    ~xyz;
    :
    :
private:
    SFLocale *cLang;
};

xyz::xyz()
{
    cLang = SFLocale::GetInstance();
    :
    :
}
```

---

**int32 SetLanguage(BDirectory \*folder, const char \*language)**

**int32 SetLanguage(BEntry \*folder, const char \*language)**

**int32 SetLanguage(BPath \*folder, const char \*language)**

**int32 SetLanguage(const char \*folder, const char \*language)**

**int32 SetLanguage(entry\_ref \*folder, const char \*language)**

Loading the language text. As first parameter the path to the "locale" files of the application is used. The second parameter is the native name of the language that should be set. It isn't the name of the "locale" file! The name of the language is saved as the string attribute "sf:language" to the file. This will be checked by "SetLanguage()" to grant the independence of the filename.

If the "locale" file is not found, the library will use the built-in default text (English) automatically.

The following results can be returned by the function:

- 0 = Ok, new language loaded and set
- 1 = Error, can't read file
- 2 = Error, file isn't a "locale" file
- 3 = Error, wrong "locale" file (application ID incorrect)
- 4 = Error, file has no language name
- 5 = Error, invalid folder

Example:

```
BPath    path;
BEntry   entry;
app_info info;
        :
        :
be_app->GetAppInfo(&info);           // get program info
entry.SetTo(&info.ref);              // get access to program
entry.GetPath(&path);                 // path + filename
path.GetParent(&path);                // get application path
path.Append("locale");               // append language folder
cLang->SetLanguage(&path, "Deutsch"); // load and set language "Deutsch"
```

---

### **void SetDefaultText(const char \*deftext[], int32 count, const char \*language)**

Set default language data. As parameter the default text field, the count of text lines and the name of the default language (mostly english) is used. The text field and the language name are internally copied by the function.

"SetDefaultText()" is normally called once at the application start up.

```
static const char      *mText[SF_LAST_ID] =
{
"Ok",                  //SFOK
"Cancel",              //SFCANCEL
"Continue",            //SFCONTINUE
        :
        :
"ERROR: There is no text to save!", //SFERRNOTEXT
"Font",                //SFFONT
"Size",                //SFSIZE
};
```

The default text should be collected in a single Include file.

Example:

```
SuperApp::SuperApp()
: BApplication("application/x-vnd.siegfriedsoft-superapp")
{
SFLocale *lang;

lang = SFLocale::GetInstance(); // get access for language data
lang->SetAppID(SF_TEXT_APP_ID);  // set applicaation ID
// --- set build-in text ---
lang->SetDefaultText(mText, SF_LAST_ID, SF_DEFAULT_LANGUAGE);
        :
        :
```

---

### **void SetAppID(const char \*app\_id)**

Setting of the application ID to identify "locale" files correctly. The string is used to identify the "locale" file that will be loaded by "SetLanguage()". It's the same ID string used at the Siegfried Locale Editor for the "locale" files. Only those "locale" files will be loaded that have the exact same string. The ID string will be copied internally by the function.

Like "SetDefaultText()" the function is to be called once at application start-up.

Example: see example of "SetDefaultText()" above.

---

---

**const char \*AppID()**

Returns the application ID set by "SetAppID ()".

---

**const char \*Language()**

Returns a pointer to the language name currently used. The name is always written in the native language. For the language german the string "Deutsch" is returned, or for french "Francais" is returned. The name of the language is set during the function call of "SetLanguage ()".

---

**const char \*Text(int32 id)**

The main function of the sfliblocale library. By a given ID number the function returns a text string of the language that was set by "SetLanguage ()". The ID number must always be  $0 \leq ID < SF\_LAST\_ID$ . If not the result NULL is returned. When the function can't find a localized text for a given (correct) ID, the built-in default text is returned automatically.

---

Example:

```
BAAlert  *alert
```

```
alert = new BAAlert("Locale", cLang->Text(SFERRNOTEXT), cLang->Text(SFCANCEL));  
alert->Go();
```

---



## Quick reference

Alphabetically ordered quick reference of all library functions.

---

### AppID()

Return the application identification of the program.

**Usage:**        `const char *add_id;  
                 app_id = AppID();`

**Parameter:**   -

**Returns:**       app\_id                Pointer to a string, the application ID set by SetDefaultText()

**Comment:**     Every application set it's own unique identifaction. The ID is included by all "locale" files. Only those files would be loaded that have the same ID as returned by the funciton.

---

### GetInstance()

Get access to the "locale" data.

**Usage:**        `SFLocale *lang;  
                 lang = SFLocale::GetInstance();`

**Parameter:**   -

**Returns:**       lang                    Pointer to the instance of the class "SFLocale"

**Comment:**     During the whole lifetime of a running application always the same instance of the class is used. The class "SFLocale" create one (and only one) instance of the class (Singleton desing pattern).

---

### Language()

Returns the native name of the current used language.

**Usage:**        `const char *land;  
                 land = Language();`

**Parameter:**   -

**Returns:**       land                    Pointer to the native name of the current language (e.g. "Deutsch", "English")

**Comment:**     -

---

### SetAppID(const char \*app\_id)

Set application ID for the program

**Usage:**        `const char *app_id = "siegfried backup locale";  
                 SetAppID(app_id);`

**Parameter:**   app\_id                String pointer to an ID

**Returns:**       -

**Comment:**     Every application has it's own unique identifier for the "locale" files. All "locale" files include the ID. By this an application can easy check if the file can be used or not. The ID is set at the application start up (before the first call of "SetLanguage()"). The ID used during the function call of "SetLanguage()". If the application ID loaded from the "locale" file is incorrect the file is rejected.

---

## SetDefaultText(const char \*mText[], int32 n, const char \*name)

Set default text and language name.

**Usage:**

```
const char *mText[SF_LAST_ID] = { ... };
inst32 n = SF_LAST_ID;
const char *name = "English";
SetDefaultText(mText, n, name);
```

**Parameter:**

mText	Default text field (table)
n	amount of text (table) lines
name	Native language name default text

**Returns:** -

**Comment:** -

---

## SetLanguage(BDirectory \*folder, const char \*language)

Set language. Load data from "locale" file.

**Usage:**

```
BDirectory *folder = new BDirectory(...);
const char *language = "Deutsch";
long ret;
ret = SetLanguage(folder, language);
```

**Parameter:**

folder	Pointer to folder where the "locale" files are.
language	Native name of the language to load and set.

**Returns:**

ret	0 = Ok, data load and set 1 = Error, can't read file 2 = Error, no "locale" file 3 = Error, wrong "locale" file (incorrect application ID) 4 = Error, file didn't include language name 5 = Error, invalid folder
-----	--

**Comment:** -

---

## SetLanguage(BEntry \*folder, const char \*language)

Set language. Load data from "locale" file.

**Usage:**

```
BEntry *folder = new BEntry(...);
const char *language = "Deutsch";
long ret;
ret = SetLanguage(folder, language);
```

**Parameter:**

folder	Pointer to folder where the "locale" files are.
language	Native name of the language to load and set.

**Returns:**

ret	0 = Ok, data load and set 1 = Error, can't read file 2 = Error, no "locale" file 3 = Error, wrong "locale" file (incorrect application ID) 4 = Error, file didn't include language name 5 = Error, invalid folder
-----	--

**Comment:** -

---

## SetLanguage(BPath \*folder, const char \*language)

Set language. Load data from "locale" file.

**Usage:**

```
BPath *folder = new BPath(...);
const char *language = "Deutsch";
long ret;
ret = SetLanguage(folder, language);
```

**Parameter:**

folder	Pointer to folder where the "locale" files are.
language	Native name of the language to load and set.

**Returns:**

ret	0 = Ok, data load and set 1 = Error, can't read file 2 = Error, no "locale" file 3 = Error, wrong "locale" file (incorrect application ID) 4 = Error, file didn't include language name 5 = Error, invalid folder
-----	--

**Comment:** -

---

## SetLanguage(const char \*folder, const char \*language)

Set language. Load data from "locale" file.

**Usage:**

```
const char *folder = "...";
const char *language = "Deutsch";
long ret;
ret = SetLanguage(folder, language);
```

**Parameter:**

folder	Pointer to folder where the "locale" files are.
language	Native name of the language to load and set.

**Returns:**

ret	0 = Ok, data load and set 1 = Error, can't read file 2 = Error, no "locale" file 3 = Error, wrong "locale" file (incorrect application ID) 4 = Error, file didn't include language name 5 = Error, invalid folder
-----	--

**Comment:** -

---

## SetLanguage(entry\_ref \*folder, const char \*language)

Set language. Load data from "locale" file.

**Usage:**

```
entry_ref *folder = ...;
const char *language = "Deutsch";
long ret;
ret = SetLanguage(folder, language);
```

**Parameter:**

folder	Pointer to folder where the "locale" files are.
language	Native name of the language to load and set.

**Returns:**

ret	0 = Ok, data load and set 1 = Error, can't read file 2 = Error, no "locale" file 3 = Error, wrong "locale" file (incorrect application ID) 4 = Error, file didn't include language name 5 = Error, invalid folder
-----	--

**Comment:** -

---

## Text(int32 id)

Returns a localized text string to a given text id.

**Usage:**     `const char *text;  
              int32 id = SF_OK;  
              text = Text(id);`

**Parameter:** id                   Identification number of a text (0 <= id < SF\_LAST\_ID)

**Returns:**    text                 Pointer to a localized text.  
                                  NULL, if id is out of range.

**Comment:**   Doesn't exist the localized text for an given id, the fuction returns a pointer to the default built-in text. If the id number is out of range NULL is returned.

---