

SIEGFRIED LOCALE LIBRARY

Entwicklerhandbuch

Copyright by Siegfried Soft

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

Allgemeines

Was ist die Siegfried Locale Library?

Die sfliblocale.so Library ermöglicht es BeOS Softwareentwicklern ihre Programme um die Fähigkeit der Mehrsprachigkeit zu erweitern. Dazu werden alle benötigten Texte eines Programms in einer externen Datei gesammelt. Die Auswertung der Datei (Daten laden, Texte ermitteln, usw.) wird von der sfliblocale.so Library durchgeführt. Anhand von ID- Nummern die für die einzelnen Texte vergeben werden, wird über die Library der jeweilige Text ermittelt und zur Verfügung gestellt.

Pro unterstützte Sprache wird eine ("locale"-)Datei mit den Texten benötigt. Die "locale"-Dateien werden mit einem eigenen Programm erzeugt, dem Siegfried Locale Editor.

Nutzungsbedingungen

Die sfliblocale.so Library, die Quelltexte sowie die Dokumentation sind Copyright by Siegfried Soft. BeOS Software Entwickler erhalten das Recht die sfliblocale.so Library und die dazugehörigen Quellcodes kostenlos in Ihre Programme zu integrieren. Dies gilt sowohl für den Einsatz im privaten Bereich als auch für die kommerzielle Nutzung und deren Weitergabe in kompilierter Form.

Gewährleistung

Siegfried Soft versichert, dass jede Anstrengung unternommen wurde, um dieses Produkt fehlerfrei zu machen. Siegfried Soft übernimmt keine Haftung für Schäden, die aus der Benutzung oder dem Missbrauch der Library und des Quellcodes entstehen.

Hinweis / Warenzeichen

Wir nehmen in diesem Handbuch auf mehrere geschützte Warenzeichen Bezug, die innerhalb des laufenden Textes nicht mehr ausdrücklich als solche gekennzeichnet sind. Aus dem Fehlen einer Kennzeichnung kann also nicht ausgeschlossen werden, dass der entsprechende Produktname frei von Rechten Dritter ist:

- Be, BeOS sind Warenzeichen der Firma Be Inc.

Systemanforderungen

Die SIEGFRIED LOCALE LIBRARY benötigt BeOS R5 oder höher. Die Library steht sowohl für die x86 Plattform als auch für die PPC Plattform zur Verfügung.

Fehlerberichte / Kritik / Anregung

Sollten Ihnen bei der Verwendung der Siegfried Locale Library Fehler auffallen oder haben Sie Anregungen für Erweiterungen: Schreiben Sie uns! Positiver und negativer Kritik, sowie Anregungen stehen wir aufgeschlossen gegenüber. Wir wollen versuchen, Ihre Ideen soweit wie möglich, in kommende Versionen der Siegfried Locale Library einfließen zu lassen. Wenn Sie Fehler finden, geben Sie bitte Ihre Rechnerkonfiguration mit an. Schreiben Sie an folgende Adresse:

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

Per Internet sind wir natürlich auch erreichbar:

WEB: <http://www.siegfried-soft.de>
EMAIL: hauff@siegfried-soft.de

Kennen Sie unser Siegfried-Backup? **Nein, dann sollten Sie dieses unbedingt nachholen.**

Siegfried Backup ist eine leistungsfähige und einfach zu bedienende Anwendung mit der jede Art von Daten und Programmen gesichert bzw. wiederhergestellt werden kann.



Es ist das ideale Programm für Einsteiger und Profis, die Ihre Daten schnell und problemlos sichern und wiederherstellen möchten. Siegfried Backup beinhaltet alle Funktionen, die Sie von einem hochwertigen Sicherungsprogramm erwarten. Darüber hinaus wurde größter Wert auf eine ausgefeilte grafische Benutzerführung gelegt.

Idee der Programmierer ist es, sowohl dem Anfänger, als auch dem Profi das Arbeiten mit Siegfried Backup so einfach und effektiv wie möglich zu gestalten.

Nicht nur beim Einstieg in die Anwendung werden Sie die vielen angenehmen Features, die Ihnen dieses Programm bietet, zu schätzen wissen. Viele Spezialfunktionen (wie z.B. Filter, Scheduler, Kompression, Add-Ons und viele andere) machen das tägliche Arbeiten mit Siegfried Backup zu einem Vergnügen.

Holen Sie sich die Demoversion von Siegfried Backup <http://www.siegfried-soft.de>

Siegfried Backup die Eintrittskarte für die optimale Datensicherung!

Arbeiten mit der sfliblocale.so Library

Dieses Kapitel enthält Informationen wie man die sfliblocale.so Library in eigene Projekte einbindet und wie die Funktionen der Library anzuwenden sind.

Das Prinzip der Lokalisierung

Die Art wie die Lokalisierung der "sfliblocale.so" Library arbeitet ist recht einfach. Jeder Text einer Anwendung der lokalisiert werden soll wird in einer Tabelle gesammelt. Identifiziert wird der Text durch seine Stellung innerhalb der Tabelle (die Tabellenzeile). Die zentrale Funktion der Library liefert einen Text anhand der übergebenen (Zeilen-)Nummer zurück. Für jede Sprache die unterstützt werden soll, wird eine solche Tabelle erzeugt. Entsprechend der Tabellenzeile wird der jeweilige Text in die gewünschte Zielsprache übersetzt und die fertige Tabelle in einer Datei abgespeichert. Um die Mehrsprachigkeit zu nutzen ist jetzt nur noch je nach gewählter Sprache die entsprechende Tabelle (Datei) zu laden und schon ist die Sprachanpassung erfolgt. Durch Hinzufügen von weiteren ("locale"-)Dateien ist die Sprachfähigkeit einer Anwendung beliebig erweiterbar, ohne jeglichen zusätzlichen Programmieraufwand!

Einbinden in ein Projekt

Zur Nutzung der Library in eigenen Projekten werden folgende Dateien benötigt:

Datei	Verzeichnis	Bemerkung
sfliblocale.so	develop/lib/x86	Locale Library für x86 Plattform
sfliblocale.so	develop/lib/ppc	Locale Library für PPC Plattform
SFLocale.h	develop/include	Klassen/Funktionsdeklarationen für die Library

Beide Dateien sind in das jeweilige Projekt hinzuzufügen.

Einbinden in eine Anwendung

Bei der sfliblocale.so Library handelt es sich nicht um eine statische Link Library, sondern um eine Library die zur Laufzeit des Programms geladen wird (Dynamic Link Library). BeOS sieht zwei Stellen vor wo solche Libraries abgelegt werden können, damit sie automatisch geladen werden. BeOS prüft zuerst ob sich im Verzeichnis der Anwendung ein Order "lib" mit der entsprechenden Library befindet. Schlägt dies fehl wird im Ordner "/boot/home/config/lib" gesucht und ggf. geladen. Wird die Library nicht gefunden erfolgt automatisch eine Fehlermeldung von BeOS.

Welcher Ort vorzuziehen ist bleibt dem Entwickler überlassen. Wir bevorzugen die erstere Variante ("lib" Verzeichnis im Ordner der Anwendung). Damit wird dem Anwender eine Deinstallation erleichtert, weil er nur das Anwendungsverzeichnis zu löschen hat ohne sich darum zu kümmern ob an anderen Stellen auch noch Teile der Anwendung zu entfernen sind.

Vorbereitung/Organisation

Damit die Library optimal genutzt werden kann, sollten alle Texte die eine Anwendung benötigt in einer zentralen (Include-)Datei gesammelt werden. Eine weitere Datei sollte die Text-IDs verwalten.

Die gesammelten Texte sind die Vorgabetexte für das Programm. Kann die Anwendung einen Text aus einer "locale"-Datei nicht ermitteln wird automatisch von der Library der Vorgabetext verwendet. Dies kann z.B. dann auftreten wenn die Anwendung erweitert wurde, die entsprechenden "locale"-Dateien aber noch nicht aktualisiert wurden. Mit der automatischen Nutzung der Vorgabetexte im Fehlerfall wird auch erreicht das die Anwendung ohne "locale"-Dateien lauffähig ist. Es wird einfach auf die internen Vorgabetexte zugegriffen. Damit die Anwendung weltweit einsatzfähig ist, ist es sinnvoll die Vorgabetexte in Englisch zu halten.

Die Namen der beiden Include-Dateien können beliebig gewählt werden. Als Beispiel wurde hier die Textdatei "SFTexte.h" und die ID-Datei "SFTextIDs.h" benannt.

Die Datei für die Texte sollte sich wie folgt aufbauen:

```
#ifndef _SFTEXTTE
#define _SFTEXTTE

#include "SFTextIDs.h"                                // Datei mit 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
```

Sobald ein neuer Text benötigt wird, wird dieser am Ende der Liste angefügt und die zugehörige Text-ID wird in "SFTextIDs.h" hinzugefügt.

Die Datei mit den Text-IDs baut sich dann wie folgt auf:

```
#ifndef _SFTEXTIDS
#define _SFTEXTIDS

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

#define MSG_SFLANGUAGE        'sflg'

//-----
// Definitionen der IDs für alle benutzten Texte
//-----

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

    SF_LAST_ID                // MUSS immer letzte ID sein!!!
};

#endif
```

Für jeden neuen Text der in der Textdatei "SFTexte.h" zur Liste hinzugefügt wird, wird in der ID-Datei "SFTextIDs.h" die entsprechende ID Kennung eingetragen. Da die Texte in "SFText.h" immer am Ende angefügt werden, ist in der ID-Datei an entsprechender Position ebenso zu verfahren (allerdings immer vor "SF_LAST_ID").

Über die Kennung `SF_LAST_ID` erhält man so automatisch die Anzahl der eingetragenen Texte.

WICHTIG: Es ist darauf zu achten, dass die Textpositionen und die Positionen der IDs immer übereinstimmen, ansonsten kommt es zu etwas seltsamen Textausgaben innerhalb der Anwendung :-).

Die ID-Datei `"SFTextIDs.h"` definiert noch zwei zusätzliche Konstanten:

`"SF_DEFAULT_LANGUAGE"` gibt den Namen der verwendeten Vorgabesprache an und das sollte eigentlich immer Englisch sein.

Weiterhin wird noch über `"SF_TEXT_APP_ID"` die Anwendungs-ID (Application ID) für die "locale"-Dateien angegeben. Eine solche Anwendungs-ID ist in jeder "locale"-Datei vorhanden. Im Siegfried Locale Editor wird diese im Feld "Application ID" aus- bzw. eingegeben. Die ID ist eine eindeutige Kennung für das Programm das "locale"-Dateien nutzt. Damit kann die Anwendung bestimmen, ob die zu ladende "locale"-Datei zugehörig ist oder nicht. Nichts bringt seltsamer Ergebnisse auf den Bildschirm als eine "locale"-Datei die gar nicht zur Anwendung gehört! :-). Die Anwendungs-ID wird vom Aufsteller der Anwendung vergeben. Alle "locale"-Dateien einer Anwendung müssen über dieselbe ID verfügen, sonst werden sie ignoriert. Wie sich die ID aufbaut ist dem Aufsteller überlassen, er kann eine beliebige Zeichenkette verwenden. Als sinnvoll erweist es sich eine Kennung zu verwenden die einen Bezug auf die jeweilige Anwendung hat. Der Siegfried Locale Editor verwendet z.B. "siegfried localeeditor locale" als ID.

Es spricht nichts dagegen als ID die Signatur des `BApplication`-Objekts der Anwendung zu verwenden, da dies hinreichend eindeutig ist.

Bereitstellung der sfiblocale Funktionalität in der eigenen Anwendung

Um die `sfiblocale.so` Library in der eigenen Anwendungen zu nutzen, sind beim Start der Anwendung bestimmte Vorbereitungen/Initialisierungen notwendig. Diese Initialisierungen werden am besten in der für die Anwendung abgeleiteten `BApplication` Klasse durchgeführt. Die Initialisierung ist nur einmal am Programmanfang durchzuführen.

Die Initialisierung hat die Aufgabe die Vorgabetexte, den Namen der Vorgabesprache und die Kennung für die "sflocale"-Dateien zu setzen. Dadurch das die Textdaten und die Text-IDs in eigene Dateien separiert wurden, ist eine Einbindung sehr einfach.

Beispiel:

```
#include "SuperApp.h"
#include "SFTextIDs.h"
#include "SFTexte.h"

int main()
{
    SuperApp *myApplication;

    myApplication = new SuperApp();           // Applikation erstellen/initialisieren
    myApplication->Run();                     // Anwendung aufrufen

    delete(myApplication);                   // Anwendung loeschen
    return(0);                               // und tschuess!!!
}

SuperApp::SuperApp()
    : BApplication("application/x-vnd.siegfriedsoft-superapp")
{
    SFLocale *lang;
    lang = SFLocale::GetInstance();          // Zugriff auf Sprachdaten
    lang->SetAppID(SF_TEXT_APP_ID);         // Anwendungskennung setzen
    // --- Vorgabetexte setzen ---
```

```
lang->SetDefaultText(mText, SF_LAST_ID, SF_DEFAULT_LANGUAGE);  
// --- Ab hier kann auf die Sprachdaten zugegriffen werden.  
// --- Die Initialisierung ist abgeschlossen.  
:  
:  
}
```

Eine Anwendung sollte sinnvoller Weise nach der Initialisierung noch die zuletzt benutzte Sprache über die Funktion "SetLanguage()" setzen.

Nachdem die Initialisierung durchgeführt wurde kann an jeder Stelle innerhalb der Anwendung auf die Textdaten zugegriffen werden. Zugriff erhält man immer über folgenden Aufruf:

```
SFLang *lang;  
lang = SFLocale::GetInstance();           // Zugriff auf Sprachdaten
```

Es bietet sich an in den Klassen die Texte benötigen das SFLocale Objekt als Klassenprivate Variable zu nutzen oder bei Basisklassen als Protected Variable zu deklarieren. Der GetInstance() Aufruf erfolgt dann in der Konstruktorfunktion der jeweiligen Klasse (siehe dazu auch Kapitel "Funktionübersicht / GetInstance()".

Nutzen der Textfunktionen

Die meistgenutzte Funktion der sfliblocale.so Library ist ohne Zweifel "Text()". Jeder Text der in einer Anwendung ausgegeben werden soll wird über diese Funktion ermittelt. Über die Funktion wird anhand der übergebenen ID-Nummer der entsprechende Text in der eingestellten Landessprache zurückgeliefert, oder ggf. der englische Vorgabetext, wenn der Text in der Landessprache nicht vorliegt.

Um z.B. einen BButton mit einem lokalisierten Label zu versehen ist nur der Zeiger auf das Textlabel durch den "Text()"-Aufruf zu ersetzen.

Beispiel:

```
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);
```

Das "SFLocale"-Object ist in dem obenstehenden Beispiel eine Klassenprivate Variable und wurde in der Konstruktorfunktion der Klasse per "GetInstance()" initialisiert. Die Text-ID "SFQUIT" ist eine benutzerdefinierte Konstante, die in der Datei "SFTextIDs.h" verwaltet werden.

Die über die Funktion "Text()" ermittelten Texte können beliebig weiterverarbeitet werden. Es ist z.B. ohne Probleme möglich an den ermittelten Text etwas anzuhängen:

Beispiel:

```
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);
```

Diese Art von Erweiterung bringt den Vorteil das sich die Anzahl der Texte die verwaltet werden müssen minimiert. In obigen Beispiel wird nur der Text "Save" verwaltet und nicht "Save" und "Save...".

Eine andere Art der Erweiterung von Texten ist das Einfügen von Zahlenwerten. Durch Nutzung der C-Library Funktion "sprintf" ist dies sehr einfach zu bewerkstelligen:

Beispiel:

```
// 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);
```

Ausgabe:

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

Wechseln der Sprache

Das Wechseln der Sprache sollte Menügesteuert erfolgen. In einem Untermenü sollten alle verfügbaren Sprachen angezeigt werden und die aktuell gewählte sollte markiert sein. Zusätzlich ist es sinnvoll auf Änderungen im Verzeichnis der Sprachdateien zu reagieren (wenn z.B. neue Dateien hinzugefügt werden).

Beispiel:

```
BMenu      *smenu;
const char  *last_language;
:
:
smenu = new BMenu(cLang->Text(SFLANGUAGE));           // Sprachmenü erstellen
// Namen der zuletzt genutzten Sprache ermitteln
last_language = GetAppPreferences(...);
// Einträge für das Menü erzeugen
BuildLanguageMenu(smenu, last_language);
cLocaleMenu = smenu;                                 // Zeiger auf Menü sichern
:
:
```

Die Funktion "BuildLanguageMenu()" übernimmt das Erzeugen des Untermenüs. Die Funktion wertet das Verzeichnis aus in dem die "locale"-Dateien untergebracht und markiert die zuletzt genutzte Sprache. Für jeden Menüeintrag wird eine BMessage mit der Kennung MSG_SFLANGUAGE erzeugt. Die BMessage enthält als zusätzliches Feld "language". Dieses Feld beinhaltet den Namen der Sprache.

Das eigentliche Wechseln der Sprache erfolgt in der abgeleiteten MessageReceive() Funktion der Fensterklasse.


```
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);    // Programminfo holen
            entry.SetTo(&info.ref);       // Entry auf Programm holen
            entry.GetPath(&path);         // Pfad + Dateiname
            path.GetParent(&path);        // Programmpfad ermitteln
            path.Append("locale");       // Sprachdaten-Pfad anhängen
            cLang->SetLanguage(&path, msg->FindString("language"));
            // Hier den neuen Name der Sprache in die Einstellungen sichern
            // SetAppPreferences(...);
            break;
        //---- Standardverarbeitung ----
        default:
            BWindow::MessageReceived(msg);
    }
}
```

Für das Wechseln der Sprache wird die Funktion "SetLanguage()" verwendet. Die Funktion benötigt den Pfad wo die "locale"-Dateien abgelegt sind und den Name der Sprache der gesetzt werden soll.

Verwalten der "locale"-Dateien

Grundprinzipiell ist es gleichgültig wo die "locale"-Dateien abgelegt werden. Die Library kann beim Laden genau feststellen ob es sich überhaupt um eine "locale"-Datei handelt und wenn ja ob die Datei auch zur Anwendung zugehörig ist.

Sinnvollerweise sollten die "locale"-Dateien für eine Anwendung in einem separaten Verzeichnis untergebracht werden. Als günstig hat es sich erwiesen diese Verzeichnis innerhalb des Hauptverzeichnis der Anwendung zu legen. Ist die Anwendung unter "/boot/home/superapp" abgelegt kann als Verzeichnis für die "locale"-Dateien "/boot/home/superapp/locale" benutzt werden. Damit hat man die Dateien zentral verwaltet und bei einer Deinstallation hat der Anwender nur das Hauptverzeichnis zu löschen, ohne sich darum Gedanken zu machen, ob noch irgendwo zusätzliche Dateien zu entfernen sind. Der Name "locale" für das Verzeichnis ist nicht zwingend, sollte aber von jedem Anwender sofort intuitiv erfasst werden.

Zur Überwachung des Verzeichnis in dem die "locale"-Dateien untergebracht sind, ist in der Konstrukturfunktion des Hauptfensters der Anwendung das Nodewatching zu aktivieren:

```
SuperAppWin::SuperAppWin(....)
    : BWindow(...)
{
    app_info  info;
    BEntry    entry;
    BPath     path;
    node_ref  nref;
    :
    :
    :
    // --- Node-watching für Locale-Verzeichnis aktivieren ---
    be_app->GetAppInfo(&info);           // Programminfo holen
    entry.SetTo(&info.ref);              // Entry auf Programm holen
    entry.GetPath(&path);                 // Pfad + Dateiname
    path.GetParent(&path);                // Programmpfad ermitteln
    path.Append("locale");                // Sprachdaten-Pfad anhängen

    if (entry.SetTo(path.Path()) == B_OK) // Zugriff auf Verzeichnis
    {
        entry.GetNodeRef(&nref);          // node_ref ermitteln
        // Überwachung aktivieren
        watch_node(&nref, B_WATCH_DIRECTORY, this);
    }
    entry.Unset();
}
```

Mit dem Nodewatching kann während des Programmlaufs einfach geprüft werden ob "locale"-Dateien aus dem Verzeichnis entfernt bzw. hinzugefügt wurden. Tritt eine Änderung auf ist das Untermenü mit den Sprachen zu aktualisieren. Das Auswerten des Nodewatching erfolgt in der abgeleiteten `MessageReceive()` Funktion der Fensterklasse.

```

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)
                {
                    //---- Eintrag aus Verzeichnis entfernt ----
                    case B_ENTRY_REMOVED:
                    //---- Eintrag verschoben -----
                    case B_ENTRY_MOVED:
                    //---- neuer Eintrag in Verzeichnis ----
                    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;
            }
            //----
            :
            :
        }
    }
}

```

Scriptfähigkeit für den Siegfried Locale Editor

Damit das Übersetzen der Texte einer Anwendung angenehm von der Hand geht, ist es sinnvoll die Vorgabetexte der Anwendung in den Siegfried Locale Editor als Vorlage zu laden/kopieren. Normalerweise würde man die Datei mit den Vorgabetexten (SFTexte.h) in den Editor übernehmen. Diese Möglichkeit steht allerdings nur dem Aufsteller der Anwendung zur Verfügung. Möchte ein Ausstehender eine Übersetzung durchführen ist dies nur möglich wenn er sich an den Aufsteller wendet und die Texte anfordert.

Mittels der Scripting-Fähigkeit von BeOS ist es aber möglich diesen Vorgang zu vereinfachen bzw. komplett zu automatisieren. Per Scripting Befehl wird es ermöglicht die aktuellen Texte der Anwendung, die ID für die "locale"-Dateien und den Namen der Vorgabesprache zu ermitteln. Der Siegfried Locale Editor nutzt diese Scripting-Fähigkeit aus und kann damit jede Anwendung die dieses Feature zur Verfügung stellt schnell und einfach auslesen ohne lästiges umherreichen von Textdateien!

Um eine Anwendung mit diesen Scripting-Fähigkeiten zu erweitern ist nur die abgeleitete BApplication Klasse geringfügig zu erweitern.

Damit die Scripting Befehl ausgewertet werden können ist die `MessageReceived()` Funktion der `BApplication` Klasse entsprechend um `B_GET_PROPERTY` zu erweitern.

```
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;
        //---- Standardverarbeitung ----
        default:
            BApplication::MessageReceived(msg);
    }
}
```

Die Anwendung wird um drei Scripting Befehle erweitert:

- "DefaultLanguage" ermittelt den Name der Vorgabesprache ("English", "Deutsch")
- "DefaultText" ermittelt ein Feld mit allen Vorgabetexten
- "TextAppID" ermittelt die Erkennungs-ID für die "locale"-Dateien der Anwendung

Die Konstanten `SF_LAST_ID`, `SF_DEFAULT_LANGUAGE` und `SF_TEXT_APP_ID` sind in den anwenderbezogenen Dateien `"SFTexte.h"` und `"SFTextIDs.h"` definiert.

Um das Scripting vollständig anzubieten müssen zusätzlich die Funktionen GetSupportedSuites() und ResolveSpecifier() eingefügt werden:

```
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);
}
```

Funktionsübersicht

SFLocale *GetInstance()

Über die Funktion "GetInstance()" wird der Zugriff auf die Textdaten hergestellt. Entgegen anderen Klassen ist das Erzeugen eines Objekts von Typ "SFLocale" nicht nötig (und auch nicht möglich). Es wird immer mit einem Zeiger auf das Objekt gearbeitet. Die Klasse "SFLocale" leitet sich von dem sog. *"Singleton Entwurfsmuster"* ab. Dadurch wird gewährleistet, dass immer nur eine Instanz der Klasse innerhalb der Anwendung verwendet wird. Dies spart Speicherplatz und hält beim Wechseln der Sprache die Daten konsistent.

Beispiel:

```
SFLocale *lang;
```

```
lang = SFLocale::GetInstance(); // Zugriff auf Textdaten herstellen
```

An jeder Stelle an der Textdaten benötigt werden kann dieser Aufruf erfolgen. Es wird immer auf dieselbe Instanz der Klasse zugegriffen. Der Zeiger auf die Instanz kann entweder als globale Variable verwendet werden, was allerdings nicht unbedingt guten Programmierstil entspricht, da das Prinzip der Kapselung durchbrochen wird. Besser ist den Zeiger auf die Instanz als "private" Variable innerhalb einer Klasse zu deklarieren (oder bei Basisklassen als "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)

Sprachdaten laden. Es wird der Pfad als erster Parameter übergeben in dem die "locale" Dateien für die Anwendung abgelegt sind. Der zweite Parameter ist die gewünschte Sprache die geladen werden soll. Dies muss nicht der Name der Sprachdatei sein! Der Name der Sprache ist als String-Attribut "sf::language" gespeichert und wird von der Laderoutine ausgewertet, daher die Unabhängigkeit von dem Dateinamen.

Kann die gewünschte "locale" Datei nicht gefunden werden, wird mit der Defaultsprache (Englisch) weitergearbeitet.

Folgende Ergebnisse können zurückgegeben werden:

- 0 = alles OK
- 1 = Fehler: Datei konnte nicht gelesen werden
- 2 = Fehler: keine Sprachdatei
- 3 = Fehler: falsche Sprachdatei (AppID nicht korrekt)
- 4 = Fehler: Datei enthält keine Länderkennung
- 5 = Fehler: ungültiges Verzeichnis

Beispiel:

```
BPath    path;
BEntry   entry;
app_info info;
:
:
be_app->GetAppInfo(&info);           // Programminfo holen
entry.SetTo(&info.ref);              // Entry auf Programm holen
entry.GetPath(&path);                // Pfad + Dateiname
path.GetParent(&path);               // Programmpfad ermitteln
path.Append("locale");              // Sprachdaten-Pfad anhaengen
cLang->SetLanguage(&path, "Deutsch"); // Sprache "Deutsch" laden
```

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

Sprachvorgabedaten setzen. Es werden die Vorgabetexte als Feld, die Anzahl der Texte und der Name der Vorgabesprache (sollte Englisch sein) übergeben. Die Vorgabetexte und der Name der Sprache werden von "SetDefaultText()" umkopiert.

Die Funktion wird üblicherweise beim Start der Anwendung aufgerufen. Das Feld der Vorgabetexte baut sich wie folgt auf:

```
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
};
```

Die Vorgabetexte sollten zentral in einer Include-Datei gesammelt werden.

Beispiel:

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

    lang = SFLocale::GetInstance(); // Zugriff auf Sprachdaten
    lang->SetAppID(SF_TEXT_APP_ID); // Anwendungskennung setzen
    // --- Vorgabetexte setzen ---
    lang->SetDefaultText(mText, SF_LAST_ID, SF_DEFAULT_LANGUAGE);
    :
    :
```

void SetAppID(const char *app_id)

Setzen der Anwendungs-ID zur korrekten identifizierung der "locale"-Dateien. Die übergebene Zeichenkette dient zur Identifizierung der "locale"-Dateien beim Laden mit "SetLanguage () ". Es muss hier dieselbe Kennung angegeben werden wie sie im Siegfried Locale Editor unter "Application ID" für die "locale"-Dateien angegeben wurde. Es werden von der Anwendung nur solche "locale"-Dateien geladen die über die exakte Kennung verfügen. Die übergebene Kennung wird von der Funktion kopiert.

Wie bei "SetDefaultText () " sollte der Aufruf von "SetAppID () " beim Starten der Anwendung ausgeführt werden.

Beispiel: siehe Beispiel bei Funktion "SetDefaultText () "

const char *AppID()

Ermitteln der gesetzten Erkennungs ID für die "locale"-Dateien der Anwendung.

const char *Language()

Ermitteln der aktuell gesetzten Sprache. Es wird ein Zeiger auf den Namen der Sprache geliefert. Der Name der Sprache ist immer in der Muttersprache verfasst. Für die Sprache Deutsch wird "Deutsch" zurückgeliefert, für Französisch "Francais". Der aktuelle Sprachname wird beim Laden einer "locale"-Datei (per "SetLanguage () " gesetzt.

const char *Text(int32 id)

Die zentrale Funktion der "sfliblocale.so" Library. Aus der übergebenen ID wird je nach geladener "locale"-Datei der entsprechende Zeiger auf den Text ermittelt. Die sollte immer $0 \leq id < SF_LAST_ID$. Ansonsten wird als Ergebnis NULL zurückgeliefert. Kann unter der angegebene ID kein Text der geladenen Sprache ermittelt werden, wird automatisch der interne (englische) Text zurückgeliefert.

Beispiel:

```
BAalert  *alert
```

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

Kurzreferenz

Alphabetisch sortierte Kurzreferenz aller sfliblocale.so Libraryfunktionen.

AppID()

Kennung der Texte für Anwendung ermitteln.

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

Parameter: -

Returns: app_id Zeiger auf Kennung.

Bemerkung: Jede Anwendung vergibt eine Kennung die in allen Sprachdateien der Anwendung gespeichert wird. Damit kann sichergestellt werden das nur die Sprachdateien geladen werden die auch zu der Anwendung gehören.

GetInstance()

Zugriff auf "locale"/Text-Daten ermöglichen.

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

Parameter: -

Returns: lang Zeiger auf Instanz der Klasse SFLocale.

Bemerkung: Während des kompletten Laufs einer Anwendung wird immer mit derselben Instanz gearbeitet. Die Klasse SFLocale erzeugt immer nur genau eine Instanz der Klasse (sog. Singleton Entwurfsmuster).

Language()

Sprache ermitteln; Landeskennung (Name) der aktuelle geladenen Sprache ermitteln.

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

Parameter: -

Returns: land Zeiger auf Länderkennung (z.B. "Deutsch", "English")

Bemerkung: -

SetAppID(const char *app_id)

Identifizierungskennung für Sprachdateien setzen.

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

Parameter: app_id Zeiger auf Kennung

Returns: -

Bemerkung: Jede Anwendung vergibt eine Kennung die in allen Sprachdateien der Anwendung gespeichert wird. Damit kann sichergestellt werden das nur die Sprachdateien geladen werden die auch zu der Anwendung gehören. Die Kennung wird an sinnvollsten beim Start der Anwendung gesetzt (auf jeden Fall vor SetLanguage). Die Kennung wird beim Laden von Sprachdaten (SetLanguage) ausgewertet. Stimmt beim Laden von Sprachdateien die ID der Sprachdatei und die gesetzte nicht überein, wird die Datei zurückgewiesen.

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

Vorgabetext und -sprache setzen.

Aufruf:

```
const char *mText[SF_LAST_ID] = { ... };
int32 n = SF_LAST_ID;
const char *name = "English";
SetDefaultText(mText, n, name);
```

Parameter:

mText	Tabelle mit Vorgabetexten
n	Anzahl der Tabellenzeilen
name	Name der Sprache

Returns: -

Bemerkung: -

SetLanguage(BDirectory *folder, const char *language)

Sprache setzen. Text für die jeweilige Sprache laden.

Aufruf:

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

Parameter:

folder	Zeiger auf das Verzeichnis in dem sich die Sprachdateien befinden.
language	Name bzw. Attribut ("sf.language") der Sprache.

Returns: ret

- 0 = alles OK
- 1 = Fehler: Datei konnte nicht gelesen werden
- 2 = Fehler: keine Sprachdatei
- 3 = Fehler: falsche Sprachdatei (AppID nicht korrekt)
- 4 = Fehler: Datei enthält keine Länderkennung
- 5 = Fehler: ungültiges Verzeichnis

Bemerkung: -

SetLanguage(BEntry *folder, const char *language)

Sprache setzen. Text für die jeweilige Sprache laden.

Aufruf:

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

Parameter:

folder	Zeiger auf das Verzeichnis in dem sich die Sprachdateien befinden.
language	Name bzw. Attribut ("sf.language") der Sprache.

Returns: ret

- 0 = alles OK
- 1 = Fehler: Datei konnte nicht gelesen werden
- 2 = Fehler: keine Sprachdatei
- 3 = Fehler: falsche Sprachdatei (AppID nicht korrekt)
- 4 = Fehler: Datei enthält keine Länderkennung
- 5 = Fehler: ungültiges Verzeichnis

Bemerkung: -

SetLanguage(BPath *folder, const char *language)

Sprache setzen. Text für die jeweilige Sprache laden.

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

Parameter: folder Zeiger auf das Verzeichnis in dem sich die Sprachdateien befinden.
 language Name bzw. Attribut ("sf:language") der Sprache.

Returns: ret 0 = alles OK
 1 = Fehler: Datei konnte nicht gelesen werden
 2 = Fehler: keine Sprachdatei
 3 = Fehler: falsche Sprachdatei (AppID nicht korrekt)
 4 = Fehler: Datei enthält keine Länderkennung
 5 = Fehler: ungültiges Verzeichnis

Bemerkung: -

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

Sprache setzen. Text für die jeweilige Sprache laden.

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

Parameter: folder Zeiger auf das Verzeichnis in dem sich die Sprachdateien befinden.
 language Name bzw. Attribut ("sf:language") der Sprache.

Returns: ret 0 = alles OK
 1 = Fehler: Datei konnte nicht gelesen werden
 2 = Fehler: keine Sprachdatei
 3 = Fehler: falsche Sprachdatei (AppID nicht korrekt)
 4 = Fehler: Datei enthält keine Länderkennung
 5 = Fehler: ungültiges Verzeichnis

Bemerkung: -

SetLanguage(entry_ref *folder, const char *language)

Sprache setzen. Text für die jeweilige Sprache laden.

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

Parameter: folder Zeiger auf das Verzeichnis in dem sich die Sprachdateien befinden.
 language Name bzw. Attribut ("sf:language") der Sprache.

Returns: ret 0 = alles OK
 1 = Fehler: Datei konnte nicht gelesen werden
 2 = Fehler: keine Sprachdatei
 3 = Fehler: falsche Sprachdatei (AppID nicht korrekt)
 4 = Fehler: Datei enthält keine Länderkennung
 5 = Fehler: ungültiges Verzeichnis

Bemerkung: -

Text(int32 id)

Text aus ID-Nummer ermitteln.

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

Parameter: `id` Identifikationsnummer des Text. ($0 \leq id < SF_LAST_ID$)

Returns: `text` Zeiger auf gewünschten Text.
 NULL, wenn Text nicht vorhanden ist.

Bemerkung: Ist die ID nicht vorhanden wird versucht den Text aus dem Vorgabetext (mit SetDefaultText gesetzt) zu ermitteln. Schlägt auch dies fehl wird NULL zurückgeliefert.
