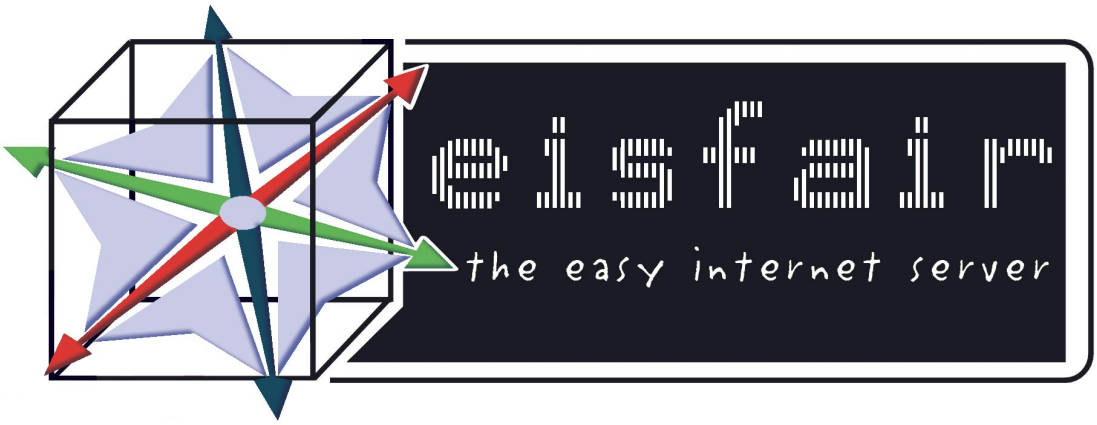


libCUI - Dokumentation

Das eisfair Dokumentations-Team

Letzte Änderung am 12. Dezember 2016

Inhaltsverzeichnis

1. Einleitung

1.1. Was ist die libCUI

Die Abkürzung CUI steht für Character User Interface. Gemeint ist damit eine bedienerfreundliche Gestaltung der Benutzeroberfläche, die im Gegensatz zum GUI (Grapical User Interface) auf einer reinen Zeichendarstellung beruht und vornehmlich für die Textkonsole gedacht ist.

Die libCUI ist eine auf curses (GNU ncurses) basierende Bibliothek, zur einfachen und schellen Entwicklung von CUI-Anwendungen. Programmiert wird dabei mit Ansichten, die in der libCUI Terminologie als Fenster bezeichnet werden.

Zentraler Bestandteil der Bibliothek ist eine Art Fenstermanager der die Darstellung und Verwaltung von Fenstern, sowie die Steuerung der Tastatur- und Mauseingabe übernimmt.

Aufbauend auf einer generischen Fensterfunktionalität, kann sich der CUI-Programmierer seine eigenen Ansichten durch die Vorgabe von Fensterstilen und durch die Implementierung von Hook-Funktionen erstellen. Daneben sind auch eine Reihe vordefinierter Fensterklassen verfügbar, die auf einfache Weise in eigene Anwendungen eingebunden werden können. Dies sind im allgemeinen Kontrollelemente wie Textfelder, Buttons und Listboxen.

Während die libCUI dem Programmierer das Update-Handling der Benutzerschnittstelle, sowie die Tastatur- und Mausbehandlung vollständig abnimmt, werden für die Textausgabe selbst die Funktionen der ncurses-Bibliothek verwendet. Es sind daher Grundkenntnisse dieser API erforderlich um erfolgreich Anwendungen mit der libCUI erstellen zu können.

Das vorliegende Dokument ist keine vollständige Referenz der CUI-API sondern versucht ein didaktischer Leitfaden zu sein, der den Programmierer schrittweise an den Umgang mit der Bibliothek heranführt. Für eine vollständige Liste aller Funktionsprototypen und Datentypen sein auf die Datei "cui.h" verwiesen, die sich unter "/usr/include" befinden sollte.

1.2. Historie

Alles begann damit, dass für eisfair ein auf curses basierendes Programm zur Darstellung des Systemmenüs entwickelt wurde. Später kamen weitere Programme, wie z.B. der Konfigurationseditor und ein Dokumenten-Viewer dazu, die auf gemeinsamen Code-Dateien beruhten um redundante Code-Pflege zu vermeiden. Dies wurde jedoch mehr und mehr zum Problem. Zum einen waren die bisherigen Dateien konzeptionell nicht dafür vorge-

sehen, um universell einsetzbar zu sein, zum anderen wurde das Projektverzeichnis durch weitere Programme immer unübersichtlicher.

Um hier eine Trennung der Projekte und zugleich eine Verallgemeinerung der zugrunde liegenden Programmroutinen zu erreichen wurde die libCUI entwickelt. Dabei wurde darauf geachtet, dass auch mit der neuen Basis die eisfair-typische Darstellung der Anwendungen weiterhin gewahrt bleibt.

Hinweis

Zum Zeitpunkt der Erstellung dieses Dokuments ist die libCUI noch sehr jung und noch nicht alle eisfair-Programme wurden bisher auf die Verwendung der Bibliothek umgestellt. Es ist noch immer damit zu rechnen, dass sich die API grundlegend ändert, weshalb in diesem Stadium dringend empfohlen wird, eigene Programme statisch mit der libCUI zu verlinken. Siehe dazu auch das Kapitel ”übersetzen von libCUI Anwendungen” (Seite 7)

1.3. Lizenz

Die libCUI unterliegt der GPL.

2. Erste Schritte mit der libCUI

2.1. ”Hallo CUI world!”

Um nicht mit der guten alten Tradition von Programmierbüchern zu brechen, soll auch diese Einführung mit dem obligatorischen ”Hallo World” Programm beginnen:

```
#include <cui.h>

void quit(void)
{
   WindowEnd();
}

int main(void)
{
   WindowStart(TRUE, TRUE);
   atexit(quit);

   MessageBox(WindowGetDesktop(),
              "Hallo World\n"
              "This is my first libcui program!",
              "Message",
              MB_OK);

   return 0;
}
```

Zunächst wird die Datei ”cui.h” eingebunden, die alle Funktionsprototypen, Datentypen und Konstanten der libCUI bekannt macht. In der ”main”-Routine wird dann über den Funktionsaufruf ”WindowStart” in den curses-Modus gewechselt, wobei die beiden Parameter sowohl den Farbmodus, als auch die Mauseingabe aktivieren.

```
   void WindowStart(int color, int mouse);
```

Anschließend wird über ”atexit” eine Exit-Routine angemeldet, die in jedem Fall dafür sorgt, dass der curses-Modus auch wieder korret über ”WindowEnd” beendet wird und alle von der libCUI verwalteten Datenstrukturen freigegeben werden. Zu guter letzt wird über ”MessageBox” ein Meldungsfenster angezeigt, das eine zweizeilige Meldung ausgibt.

Das Programm mogelt sich zugegebenermaßen um die eigentliche Fensterprogrammierung herum, indem die fertige Dialogklasse ”MessageBox” verwendet wird um Text auszugeben.

Zu einem späteren Zeitpunkt wird darauf jedoch noch wesentlich ausführlicher eingegangen.

2.2. übersetzen von libCUI Anwendungen

Damit ein libCUI-Programm lauffähig ist, muss es mit der Biblithek bei der übersetzung verknüpft (gelinkt) werden. Dies funktioniert sebstverständlich nur dann, wenn sich die Entwicklungsdateien (headers and libraries) der libCUI auf dem Build-System befinden.

Das Linken kann auf zwei Arten erfolgen: dynamisch und statisch. Bei der dynamischen Verknüpfung werden lediglich Verweise auf eine gemeinsame Bibliotheksdatei in dem erzeugten Programm hinterlegt. Diese müssen später vom Programm-Loader des Betriebsystems aufgelöst werden, bevor das Programm zur Ausführung kommen kann. Der Vorteil dabei ist, dass alle Programme eine gemeinsame Bibliothek verwenden und damit Resourcen schonen. Zudem kommen bei einem Update der Bibliothek alle Programme zugleich in den Genuss von Fehlerkorrekturen, solange sich das Interface der Bibliothek nicht verändert hat.

Beim statischen Linken wird der verwendete Code der Bibliothek in das Programm übernommen, wodurch deutlich größere Binärdateien erzeugt werden. Diese haben dann aber den Vorteil, dass sie von konzeptionellen Änderungen der Bibliothek unabhängig sind und auch auf Systemen funktionieren, auf denen die libCUI nicht als gemeinsame Datei vorliegt. Zum momentanen Zeitpunkt wird zum statischen Linken geraten, da sich die libCUI noch in einem recht frühen Entwicklungsstadium befindet.

2.2.1. Dynamisches Linken

```
gcc -Wall -Wstrict-prototypes -o hallo.o hallo.c
gcc -o hallo hallo.o -lcui
```

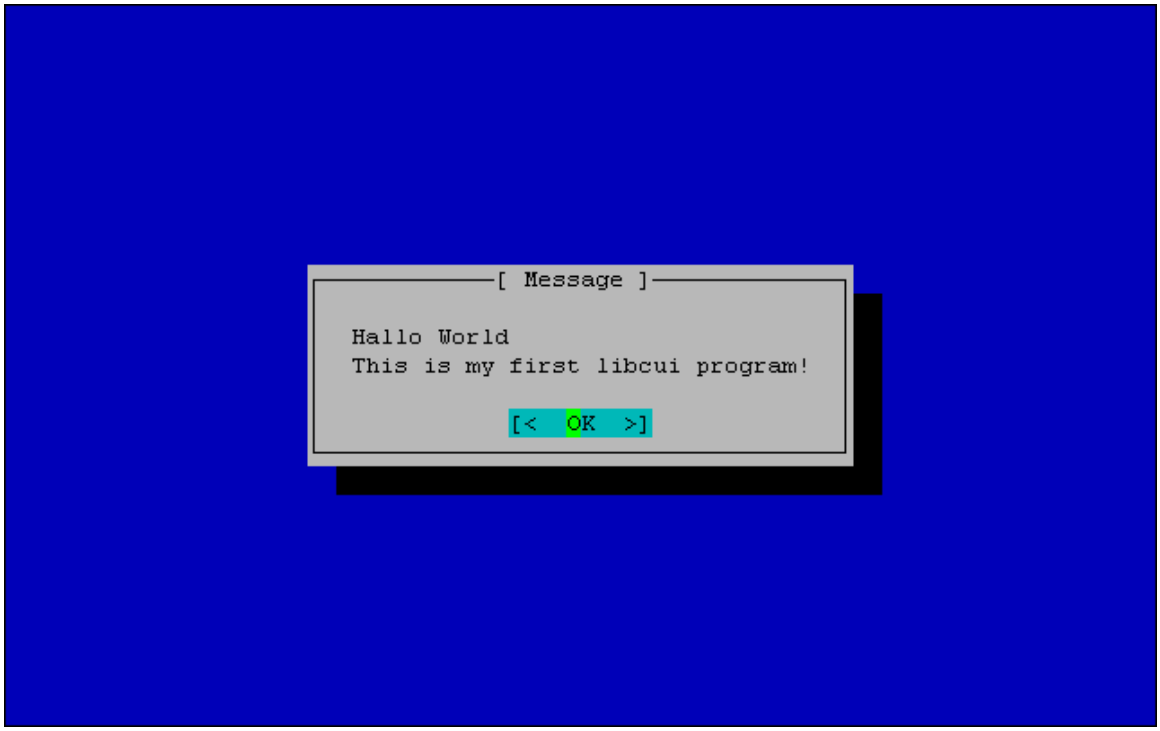
Hier wird in zwei Schritten zunächst die Datei hallo.c übersetzt und anschließend mit der Bibliothek libCUI zu einem ausführbaren Programm verlinkt. Bei nur einer Code-Datei kann dies auch in einem einzelnen Schritt erfolgen:

```
gcc -Wall -Wstrict-prototypes -o hallo hallo.c -lcui
```

2.2.2. Statisches Linken

```
gcc -Wall -Wstrict-prototypes -o hallo.o hallo.c
gcc -o hallo hallo.o /usr/lib/libcui.a -lcurses
```

Hier wird in zwei Schritten zunächst die Datei hallo.c übersetzt und anschließend mit der Bibliothek libCUI zu einem ausführbaren Programm verlinkt. Dabei wird die statische Bibliotheksdatei "/usr/lib/libcui.a" einfach in die Liste der zu verlinkenden Module aufgenommen wodurch sie vom Linker fest eingebunden. Zu beachten ist jetzt allerdings, dass zur Auflösung der curses-Funktionen, die curses-Library ebenfalls angegeben werden muss.



Bei nur einer Code-Datei kann das auch in einem einzelnen Schritt erfolgen:

```
gcc -Wall -Wstrict-prototypes -o hallo hallo.c /usr/lib/libcui.a \
    -lcurses
```

2.3. Das Ergebnis

Egal ob nun statisch oder dynamisch gelinkt wurde, sollte sich nach dem obigen Beispiel eine Datei ”hallo” im aktuellen Verzeichnis befinden, die sich mit

```
eis # ./hallo
```

ausführen lässt. Das Ergebnis ist ein Programm, dass die Meldung ”Hallo World”, ”This is my first libcui program!” in einem Meldungsfenster vor einem (blauen) Hintergrund ausgibt. Das Meldungsfenster besitzt eine Schaltfäche über die das Fenster entweder mit der Leertaste oder der Enter-Taste geschlossen werden kann. Falls die Konsole die Maus unterstützt, kann das Fenster auch mit diesem Eingabegerät geschlossen werden. Zudem lässt sich die MessageBox durch Ziehen der Maus auf dem Fensterrahmen im sichtbaren Bereich verschieben.

Wird die Größe des Konsolenfensters geändert, wird das Meldungsfenster in der Fenstermitte zentriert.

2.4. Allgemeine Hinweise

2.4.1. Koordinatensysteme

Vergleichbar mit gängigen Fenstersystemen wird bei Koordinatenangaben innerhalb der libCUI eine x, y-Reihenfolge eingehalten. Demgegenüber nennt die curses- Bibliothek die y-Koordinate zuerst, also y, x. Dies mag zu einiger Verwirrung führen, denn immer wenn mit curses-Routinen zur Textausgabe gearbeitet wird, ist dieser Unterschied zu beachten.

Um die Programmierung in dieser Hinsicht zu erleichtern wird die libCUI zu einem späteren Zeitpunkt Makro-Definitionen erhalten, die die Curses-Routinen so erscheinen lassen, als seien sie innerhalb der libCUI definiert. Dabei werden die Koordinaten entsprechend getauscht werden.

2.4.2. Mausunterstützung

Damit die Maus auf der Textkonsole funktioniert, muss die zugrundeliegende curses- Bibliothek mit Mausunterstützung (z.B. für den gpm-Mouse Daemon) übersetzt worden sein. Dies ist oftmals nicht der Fall. In Terminal-Fenstern unter X11 (KDE- / Gnome- Terminal), sowie im ssh Client PuTTY ist die Mausunterstützung i.d.R enthalten. Allerdings ist es dem Autor nicht gelungen, auch die Mouse-Move Ereignisse sichtbar zu machen. Diese Eigenschaft muss in den Tiefen der terminfo-Dateien verborgen sein. Sachdienliche Hinweise sind stets willkommen... :-)

3. Elementare Fenstertechnik

3.1. Ein wenig Theorie zur Fenstertechnik

3.1.1. Was ist ein Fenster?

Ein Fenster innerhalb der libCUI beschreibt einen rechteckigen Bildschirmbereich, der mit einer rudimentären Basisfunktionalität ausgestattet ist. Um ein Fenster an eine spezielle Aufgabe anzupassen wird es mit Eigenschaften und Hook-Funktionen versehen.

Eigenschaften sind u.a. Fensterstile und Farben. Sie steuern hauptsächlich das optische Erscheinungsbild des Fensters und dessen Verhalten innerhalb des Fensterstapels.

über Hook-Funktionen werden dem Fenster vom Fenstermanager Nachrichten zugesendet, auf die es durch selbstimplementierte Weise reagieren kann. Solche Nachrichten sind z.B. die Aufforderung zum Zeichnen des Fensterbereichs oder die übergabe von Tastatureingaben an das Fenster.

über einen nichttypisierten Datenzeiger kann ein Fenster Instanz-Daten verwalten, die es für seine Funktion benötigt.

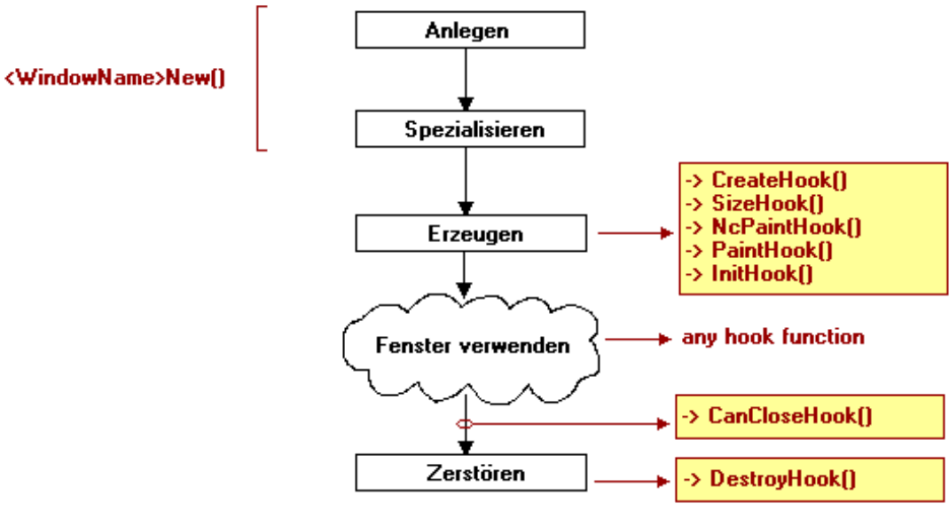
3.1.2. Lebenszyklus

Ein Fenster macht üblicherweise die folgenden Stadien im Laufe seiner Existenz durch: Anlegen, Spezialisieren, Erzeugen und Zerstören.

Anlegen:
Das Anlegen einer neuen Fensterinstanz erfolgt mit Hilfe der Funktion "WindowNew". Dabei werden dem Fenster eine Position, eine Vater-Beziehung und eine Reihe von Fensterstilen mitgegeben. Das Ergebnis ist eine Struktur vom Typ CUIWINDOW, über die das Fenster in Zukunft angesprochen und identifiziert werden kann.

Spezialisieren:
Durch das Spezialisieren bekommt das Fenster weitere Eigenschaften (z.B. Farben), seine Funktionalität (Hook-Funktionen) und seine Instanzdaten beigebracht. Zu diesem Zweck gibt es eine Reihe von API-Funktionen. Siehe dazu u.a. den Abschnitt "Hook Funktionen".

Erzeugen:
Bevor ein Fenster im Fensterstapel sichtbar werden kann, muss es erzeugt werden. Dies erfolgt mit der Funktion "WindowCreate". Dabei werden die erforderlichen Curses-Strukturen erzeugt und das Fenster in den Fensterstapel eingebunden.



Zerstören:
Zerstört wird ein Fenster über die Funktion "WindowDestroy". Damit wird das Fenster und alle seine Kindfenster aus dem Fensterstapel entfernt. Zudem werden die zugehörigen Datenstrukturen freigegeben. Das heißt auch, dass ein Fenster wieder neu angelegt werden muss, bevor es erneut verwendet werden kann.

Es bietet sich an, das Anlegen und Spezialisieren in einer Funktion zu vereinen. Eingebürgert hat sich dabei die Namenskonvention *Fenstername*New. Beispiele: "EditNew", "LabelNew", "ListboxNew" ...

3.1.3. Fensterhierarchie

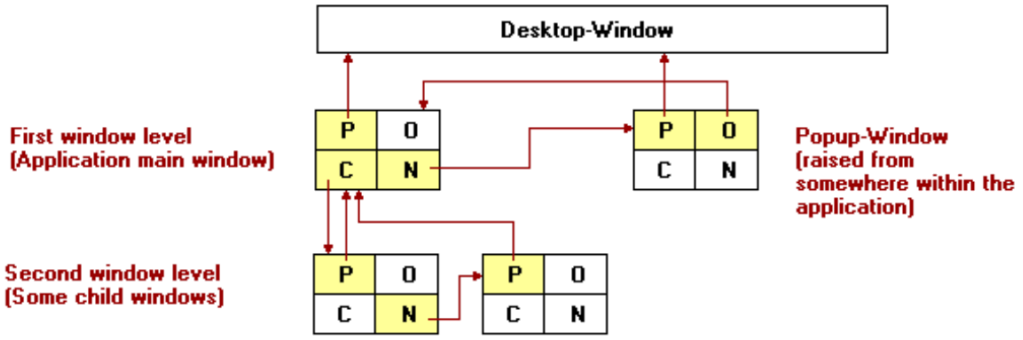
Grob unterteilt gibt es in der libCUI zwei Fensterkategorien: Popup-Fenster und Kindfenster.

Kindfenster sind immer ihrem Vaterfenster untergeordnet. Die Positionierung erfolgt relativ zum Vaterfenster und der sichtbare Bereich wird durch den Client Bereich des Vaterfensters gegrenzt (Clipping).

Popup-Fenster dagegen sind immer dem Desktopfenster untergeordnet und besitzen lediglich einen Besitzer- (Owner-) Verweis auf dasjenige Popup-Fenster aus dessen Kontext (Kindfenster) heraus das Vaterfenster angegeben wurde. Der sichtbare Bereich wird lediglich von weiteren übergeordneten Popup-Fenstern maskiert, nicht jedoch vom Vaterfenster.

Fenster werden in einem Stapel hierarchisch angeordnet, wobei die Basis aller Fenster das Desktopfenster ist. Dieses existiert bereits nach dem Aufruf von "WindowStart()", besitzt jedoch keine besondere Funktion (außer der Darstellung einer einfarbigen Fläche). über dem Desktop ist der Fensterstapel angeordnet, wobei jedes Fenster vier Referenzen kennt: Das Vaterfenster (P = Parent), das Besitzerfenster (O = Owner), das nächste Fenster in der selben Hierarchieebene (N = Next) und das erste Fenster in der Liste der Kindfenster (C = Child). Es ergibt sich die in der Abbildung dargestellte Struktur.

Wird ein Fenster aus dem Fensterstapel entfernt ("WindowDestroy"), dann verschwinden



auch sämtliche Kindfenster von der Anzeige. Das selbe passiert auch dann, wenn die Sichtbarkeit eines Fensters beeinflusst wird.

Kindfenster des Desktops sind übrigens immer Popup-Fenster. Dieser Stil wird beim Erzeugen erzwugen, wenn als Vaterfenster das Desktopfenster angegeben wurde.

3.1.4. Bereiche eines Fensters

Ein Fenster in der libCUI besitzt zwei Darstellungsbereiche: Den Client Bereich und den Nicht-Client Bereich. Im Nicht-Client Bereich werden Fensterdekorationen wie z.B. ein Rahmen eine Titlezeile oder Bildlaufleisten dargestellt. Im Client Bereich kann ein Fenster seine Daten zur Ansicht bringen oder weitere Kindfenster anordnen. Jedes Fenster besitzt immer beide Bereiche. Ob der Nicht-Client Bereich tatsächlich sichtbar ist, wird über Fensterstile gesteuert.

3.1.5. Beispiel

Anhand eines Code-Beispiels werden nun einige der vorgestellten Konzepte erläutert. Das vorgestellte Programm soll ein einfaches Popup-Fenster (vergleichbar dem Hallo-World Programm aus dem vorigen Kapitel) zentriert darstellen, in dem die aktuelle Systemzeit ausgegeben wird. Hier das Programmlisting:

```
#include <stdlib.h>
#include <time.h>
#include <cui.h>

#define MY_TIMER 100

void quit(void)
{
   WindowEnd();
}
```

```
void MyPaintHook(void* w)
{
   CUIWINDOW* win = (CUIWINDOW*) w;
   CUIRECT    rc;
   char       tstr[64];
   time_t     ti;
   struct tm* loctime;

   WindowGetClientRect(win, &rc);

   time(&ti);

   loctime = localtime(&ti);

   sprintf(tstr, "%02i:%02i:%02i",
      loctime->tm_hour,
      loctime->tm_min,
      loctime->tm_sec);
   mvwaddstr(win->Win, rc.H / 2 - 1,
            (rc.W - strlen(tstr)) / 2, tstr);

   strcpy(tstr, "F10 = Close");
   mvwaddstr(win->Win, rc.H / 2 + 1,
            (rc.W - strlen(tstr)) / 2, tstr);
}

int MyKeyHook(void* w, int key)
{
   if (key == KEY_F(10))
   {
      if (WindowClose((CUIWINDOW*) w, EXIT_SUCCESS))
      {
         WindowKillTimer((CUIWINDOW*) w, MY_TIMER);
      }

      return TRUE;
   }
   return FALSE;
}

void MyTimerHook(void* w, int id)
{
   WindowInvalidate((CUIWINDOW*) w);
}

int main(void)
{
```

```
    CUIWINDOW* mywin;

    WindowStart(TRUE, TRUE);
    atexit(quit);

    mywin = WindowNew(WindowGetDesktop(),
                      0, 0, 25, 7,
                      CWS_POPUP | CWS_BORDER | CWS_CENTERED);

    WindowSetText       (mywin, "Clock Window");
    WindowSetPaintHook  (mywin, MyPaintHook);
    WindowSetKeyHook    (mywin, MyKeyHook);
    WindowSetTimerHook  (mywin, MyTimerHook);

    WindowCreate        (mywin);
    WindowSetTimer      (mywin, MY_TIMER, 500);

    return WindowRun();
}
```

Zunächst ist die ”main”-Routine zu beachten. Hier wird wie gehabt der curses-Modus aktiviert und anschließend über ”WindowNew” eine neue Fensterinstanz angelegt. Diese erhält einen Verweis auf das Vaterfenster (hier das Desktop-Fenster), eine Position (0, 0) und eine Größe (25, 7). Zudem werden mehrere Fensterstile angegeben, die festlegen, dass das Fenster ein Popup-Fenster mit einem Rand sein soll und immer zentriert in der Bildschirmmitte anzuordnen ist.

Nun wird das Fenster spezialisiert, indem es einen Fenstertitel und drei Hook-Funktionen zugewiesen bekommt. Die Hook-Funktion ”MyPaintHook” ist für das Zeichnen des Client-Bereichs des Fensters zuständig, die Hook-Funktion ”MyKeyHook” bearbeitet die Tastatureingaben des Anwenders und die Funktion ”MyTimerHook” empfängt Timer-Ereignisse, falls ein Window-Timer aktiviert wurde.

Abschließend wird das Fenster über ”WindowCreate” erzeugt und erhält zudem durch ”WindowSetTimer” einen Window-Timer zugewiesen, der alle 500 Millisekunden abläuft.

Die Funktion ”WindowRun” aktiviert nun den Window-Manager und kehrt erst dann zum Hauptprogramm zurück, wenn das Fenster geschlossen wurde. Der Rückgabewert ist dabei der Exit-Code, der der WindowClose-Funktion mitgegeben wurde (siehe ”MyKeyHook”).

Die Hook-Funktion ”MyPaintHook” ermittelt das Rechteck des Client-Bereichs des Fensters und zudem die aktuelle Systemzeit. Letztere wird in eine Zeichenkette umgewandelt und mit der curses-Funktion ”mvwaddstr” zentriert im Fenster ausgegeben. Dabei ist unbedingt zu beachten, dass keine curses-Funktion aufgerufen wird, die für die Aktualisierung des Bildschirminhalts zuständig ist (”wupdate” o.ä.). Die Aktualisierung wird vollständig vom Fenster-Manager übernommen.

Bei der Funktion ”MyKeyHook” handelt es sich um eine Hook-Funktion, die einen bool-Wert als Rückgabewert kennt. Über diesen wird dem System signalisiert, ob das an die

Funktion übergebene Zeichen behandelt wurde oder nicht. Falls die Funktion "FALSE" zurückgibt, wird die Standardbehandlung der libCUI für die eingegebene Taste aktiv.

Falls die Funktion "MyKeyHook" die F10-Taste erkennt, wird versucht das Fenster über die Funktion "WindowClose" zu schließen, was im vorliegenden Fall auch immer gelingt. Im Hintergrund wird dabei die hier nicht implementierte Hook-Funktion "CanCloseHook" für das aktuelle Fenster (und alle seine Kindfenster) aufgerufen, über die angefragt wird, ob das Schließen des Fensters erlaubt ist oder nicht. Falls das Fenster geschlossen wurde, wird auch der Window-Timer beendet.

Die Timer-Funktion "MyTimerHook" verzichtet auf die Auswertung des Parameters "id", da nur ein Timer aktiviert wurde. Anderenfalls muss über "id" der richtige Timer zugeordnet werden. Der Aufruf der Funktion "WindowInvalidate" bewirkt ein Neuzeichnen des Fensters und damit das Aktualisieren des Bildschirminhaltes. Alternativ hätten auch direkt hier die Textausgaben in das Fenster erfolgen können. Allerdings darf dann am Ende der Funktion der Funktionsaufruf "WindowInvalidateScreen" nicht fehlen, der die Bildschirmausgabe (ohne Neuzeichnen des Fensters) erzwingt.

3.2. Fensterklassen und Instanzen

Im Grunde ist die Programmierung unter der libCUI in Ansätzen mit der objektorientierten Programmierung vergleichbar. Der vorliegende Abschnitt soll diese Sichtweise verdeutlichen.

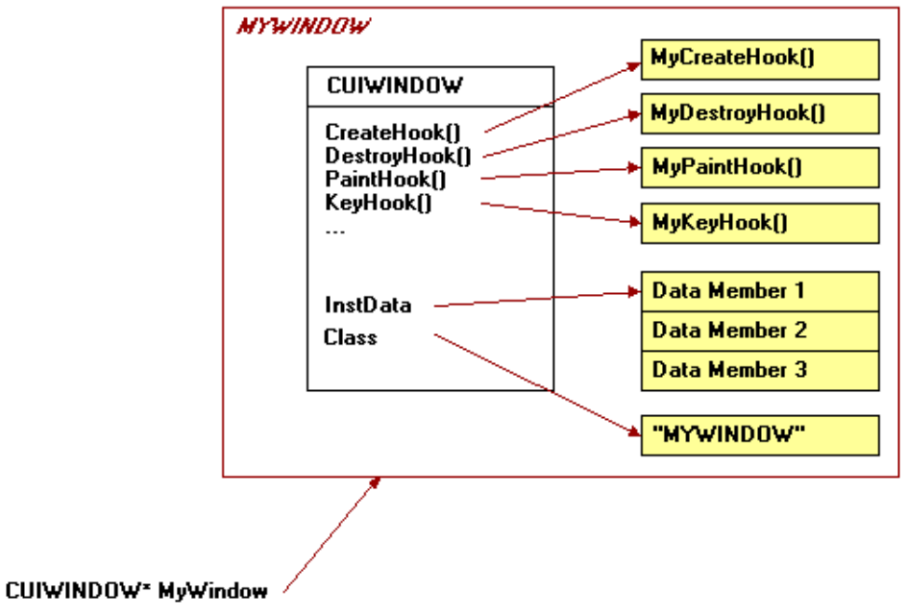
Jedes Fenster ist eine Instanz des Objekts CUIWINDOW, wobei die (virtuellen) Objektmethoden des Objekts die Hook-Funktionen sind und die Datenelemente (Objektvariablen) in CUIWINDOW->InstData gespeichert werden. Soll ein Fenster mit einem Verhalten implementiert werden, das vom Standardverhalten abweicht, was so gut wie immer der Fall sein sollte, dann erbt das Fenster zunächst das Standardverhalten, kann dieses aber durch überschreiben der Hook-Funktionen ändern.

Da jedoch unter 'C' die erforderlichen Sprachelemente fehlen, also keine Konstruktoren oder Destruktoren verfügbar sind und auch kein "this" Zeiger funktioniert, sind ein paar Klimmzüge bei der Programmierung erforderlich:

3.2.1. Objekt anlegen

Instanz anlegen -> WindowNew()
Objektmethoden zuweisen -> WindowSetHookXXXX()
Datenbereich des Objekts -> win->InstData = malloc(sizeof(MYDATA));
Curses Fenster erzeugen -> WindowCreate()

Der Datenbereich des Objekts könnte auch im Hook "WindowCreateHook" angelegt werden (den man als Konstruktor verstehen könnte).



3.2.2. Objekt löschen

Wird WindowDestroy(win) aufgerufen, dann wird das Fenster und alle Kindfenster gelöscht. Dabei wird der Hook "WindowDestroyHook" aufgerufen. Der WindowDestroyHook ist quasi der Destruktor der Objektinstanz. Hier müssen alle Datenelemente in win->InstData gelöscht werden. Im einfachsten Fall ist das:

free(win->InstData);

Das Löschen der Fensterstruktur selbst (die mit WindowNew angelegt wurde) übernimmt der Window-Manager.

3.2.3. Objekt referenzieren

Da es in 'C' keinen "this" Zeiger gibt, ist nicht bekannt, mit welcher Fensterinstanz ein Hook-Aufruf verbunden ist. Beispielsweise verwenden alle Edit-Controls die selben Hooks.

Deshalb ist im Hook immer der Parameter "w" enthalten, der ein nicht typisierter Zeiger auf die Fenster-Instanz ist (also sozusagen "this").

über w->InstData kann dann auf die Objektvariablen der Objektinstanz zugegriffen werden. Solange nicht sicher ist, dass es nur eine Instanz eines Fensters geben wird, sollte also nicht mit globalen Veriablen sondern mit Instanzdaten geabreitet werden.

3.2.4. Typsicherheit

Da immer über Zeiger vom Typ "CUIWINDOW" auf Fensterinstanzen zugegriffen wird, ist die Typsicherheit gleich null. Wird über den InstData-Zeiger auf Datenelemente zugegriffen, die diese Fensterinstanz nicht kennt, dann sind die Folgen fatal. So kann z.B. ein CUIWINDOW-Zeiger auf eine ListBox an eine Routine übergeben werden, die ein Edit-Control erwartet. Ohne weitere Maßnahmen würde das Programm mit ziemlicher Sicherheit abstürzen.

Aus diesem Grund kennt die CUIWINDOW-Struktur den "Class"-Zeiger, der auf eine konstante Zeichenkette zeigt, die den Namen der Klasse enthält. Standardmäßig zeigt dieser Zeiger immer auf die Zeichenkette "WINDOW". Dies kann jedoch nach dem Anlegen einer Fensterinstanz geändert werden.

Vor dem Zugriff auf ein Datenfeld einer Instanz, sollte deren Klassenzugehörigkeit geprüft werden. Allerdings gibt es bislang noch keinen Mechanismus innerhalb der libCUI, der doppelte Klassennamen verhindert.

Beispiel:

```
CUIWINDOW*
EditNew(CUIWINDOW* parent, const char* text,
        int x, int y, int w, int h,
        int len, int id, int sflags, int cflags)
{
   if (parent)
   {
      CUIWINDOW* edit;
      int flags = sflags | CWS_TABSTOP;
      flags &= ~(cflags);

      edit = WindowNew(parent, x, y, w, h, flags);
      edit->Class = "EDIT";

      ...

      return edit;
   }
   return NULL;
}


void
EditSetText(CUIWINDOW* win, const char* text)
{
   if (win && (strcmp(win->Class, "EDIT") == 0))
   {
```

```
      EDITDATA* data = (EDITDATA*) win->InstData;
      strncpy(data->EditText, text, data->Len);
      data->EditText[data->Len] = 0;

      if (win->IsCreated)
      {
         WindowInvalidate(win);
      }
   }
}
```

Das Code-Fragment zeigt einen Ausschnitt aus der Implementierung des Edit- Kontrollelements, das von der libCUI zur Verfügung gestellt wird. Es ist dabei zu sehen, wie in der Funktion "EditNew" neue Instanzen angelegt und mit dem Klassenname "EDIT" versehen werden. Beim späteren Zugriff auf eine Fensterinstanz mittels "EditSetText" wird der Klassenname geprüft, bevor auf die Daten des Kontrollelements zugegriffen wird.

3.3. Fensterstile

Fensterstile werden bei dem Anlegen neuer Fensterinstanzen durch "WindowNew" an die Fenster weitergegeben. Die Verknüpfung der einzelnen Stile erfolgt dabei mit dem "Oder"-Operator "|".

In der Regel beeinflusst ein Fensterstil eine Flag-Variable innerhalb der Struktur "CUIWINDOW". Der Stil "CWS_HIDDEN" bilded sich so z.B. in dem Flag "IsHidden" ab. Manche der Stile (Flags) können und dürfen über geeignete API- Funktionen während der Existenz des Fensters geändert werden (z.B. WindowShow()), die überigen, für die keine API- Funktionen zur Verfügung stehen, sollten dagegen so beibehalten werden, wie sie von "WindowNew" übernommen wurden.

Die folgende Tabelle zeigt eine übersicht über die Fensterstile, die in der libCUI bisher definiert und mit einer Funktion belegt sind:

Stil	Bedeutung
CWS_NONE	Platzhalter für "keine Stilangabe".
CWS_BORDER	Fenster hat einen Rand.
CWS_CAPTION	Fenster hat eine Kopfzeile.
CWS_HIDDEN	Das Fenster ist verborgen.
CWS_DISABLED	Das Fenster wird deaktiviert dargestellt und erlaubt keine Tastatureingabe.
CWS_TABSTOP	Das Fenster akzeptiert den Eingabefokus.
CWS_CENTERED	Das Fenster wird immer zentriert dargestellt (nur sinnvoll in Kombination mit CWS_POPUP).
CWS_POPUP	Das Fenster ist ein Popup-Fenster.
CWS_MAXIMIZED	Das Fenster ist maximiert und wird in seiner Größe immer an den Client-Bereich des Vaterfensters (bzw. des Desktop-Fensters) angepasst.
CWS_MINIMIZED	Das Fenster ist minimiert (im Effekt wie CWS_HIDDEN).
CWS_STATUSBAR	Das Fenster hat eine Statuszeile.
CWS_DEFOK	Das Fenster ist das Standard-Kontrollelement der ENTER-Taste. Falls ENTER nicht mit einer speziellen Funktion belegt ist, wird das Kindfenster mit dem Stil CWS_DEFOK gesucht. Handelt es sich dabei um eine Schaltfläche, dann wird diese betätigt.
CWS_DEFCANCEL	Wie CWS_DEFOK, nur dass hier die ESCAPE-Taste mit dem Kontrollelement verbunden ist.

Die Flags "CWS_MAXIMIZED", "CWS_MINIMIZED", "CWS_HIDDEN" und auch "CWS_-DISABLED" können mit den folgenden API-Funktionen während der Existenz des Fensters beeinflusst werden:

```
int  WindowMaximize(CUIWINDOW* win, int state);
int  WindowMinimize(CUIWINDOW* win, int state);
void WindowHide(CUIWINDOW* win, int state);
void WindowEnable(CUIWINDOW* win, int state);
```

3.4. Hook-Funktionen

Wie bereits in den vorhergehenden Abschnitten mehrfach angedeutet, wird das Verhalten eines Fensters von den Hook-Funktionen bestimmt. Hook-Funktionen werden vom Fenster-Manager aufgerufen, wenn ein Ereignis an ein Fenster gesendet werden soll. Ist dabei eine Hookfunktion nicht zugewiesen, dann tritt die jeweilige Standard-Behandlung in Kraft.

Die folgenden Funktionen werden verwendet um einem Fenster Hook-Funktionen zuzuweisen:

```
void WindowSetCreateHook(CUIWINDOW* win, HookProc proc);
void WindowSetInitHook(CUIWINDOW* win, HookProc proc);
void WindowSetCanCloseHook(CUIWINDOW* win, BoolHookProc proc);
```

```
void WindowSetDestroyHook(CUIWINDOW* win, HookProc proc);

void WindowSetPaintHook(CUIWINDOW* win, HookProc proc);
void WindowSetNcPaintHook(CUIWINDOW* win, Hook2IntProc proc);
void WindowSetSizeHook(CUIWINDOW* win, BoolHookProc proc);
void WindowSetSetFocusHook(CUIWINDOW* win, Hook1PtrProc proc);
void WindowSetKillFocusHook(CUIWINDOW* win, HookProc proc);

void WindowSetKeyHook(CUIWINDOW* win, BoolHook1IntProc proc);

void WindowSetMMoveHook(CUIWINDOW* win, Hook2IntProc proc);
void WindowSetMButtonHook(CUIWINDOW* win, Hook3IntProc proc);
void WindowSetVScrollHook(CUIWINDOW* win, Hook2IntProc proc);
void WindowSetHScrollHook(CUIWINDOW* win, Hook2IntProc proc);

void WindowSetTimerHook(CUIWINDOW* win, Hook1IntProc proc);
```

3.4.1. CreateHook

Prototyp:

```
void CreateHook(void* win);
```

Der Create-Hook wird immer unmittelbar nach dem Erzeugen eines Fensters (innerhalb von ”WindowCreate”) aufgerufen. Es ist die erste Hook-Funktion die im Laufe eines Lebenszyklus eines Fensters aufgerufen wird.

Der Create-Hook kann verwendet werden, um die Instanz-Daten anzulegen oder um Kindfenster zu erzeugen.

3.4.2. InitHook

Prototyp:

```
void InitHook(void* win);
```

Der Init-Hook wird ebenfalls aus der Funktion ”WindowCreate” heraus aufgerufen. Allerdings nachdem der CreateHook ausgeführt und das Fenster über den PaintHook (und den NcPaintHook) zum Neuzeichnen aufgefordert wurde.

Der Init-Hook kann verwendet werden, um z.B. einen Anmeldedialog oder Ähnliches darzustellen, der als Popup-Fenster über dem gerade erzeugten Fenster erscheinen soll.

3.4.3. CanCloseHook

Prototyp:

```
int CanCloseHook(void* win);
```

Soll ein Fenster über die API-Funktion "WindowClose" geschlossen werden, dann fragt der Fenster- Manager das betroffene Fenster und alle seine Kindfenster an, ob ein Schließen möglich ist. Die Hook- Funktion soll dabei den Wert "TRUE" zurückliefern, wenn das Fenster geschlossen werden kann und "FALSE" falls nicht. Als Standard für diese Hook-Funktion wird ein Rückgabewert von "TRUE" angenommen.

Der CanClose-Hook kann u.a. verwendet werden, um z.B. das Schließen eines Fensters zu verhindern, solange nicht alle Daten gespeichert wurden.

3.4.4. DestroyHook

Prototyp:

```
void DestroyHook(void* win);}
```

Der Destroy-Hook wird aufgerufen, bevor ein Fenster endgültig aus dem Fensterstapel entfernt wird. Es ist die letzt Hook-Funktion, die ein Fenster im Laufe seiner Existenz sieht.

Hier werden üblicherweise die Instanz-Daten freigegeben, die mit dem Fenster verbunden sind. Kindfenster oder die CUIWINDOW- Struktur selbst brauchen und dürfen jedoch nicht freigegeben werden. Dafür sorgt das Fenstersystem selbst.

3.4.5. PaintHook

Prototyp:

```
void PaintHook(void* win);
```

Der Paint-Hook fordert das Fenster zum Neuzeichnen des Client- Bereichs auf. Dies geschieht immer dann, wenn das Fenster erzeugt wurde, wenn sich die Größe des Fensters verändert hat oder wenn das Fenster manuell über "WindowInvalidate" als ungültig erklärt wurde.

Zur Textausgabe werden die bekannten curses-Funktionen wie "waddstr" o.ä. verwendet, wobei das zugehörige curses-Fenster über das CUIWINDOW-Datenelement "Win" angesprochen wird!

3.4.6. NcPaintHook

Prototyp:

```
void NcPaintHook(void* win, size x, size y);
```

Der NcPaint-Hook fordert das Fenster zum Neuzeichnen des Nicht- Client Bereichs (= Fensterrahmen, Titel, Bildlaufleisten etc.) auf. Dies geschieht immer dann, wenn das Fenster erzeugt wurde, wenn sich die Größe des Fensters verändert hat oder wenn das Fenster manuell über "WindowInvalidate" als ungültig erklärt wurde.

Achtung: Die libCUI implementiert bereits ein Standardverhalten zum Zeichnen des Nicht-Client Bereichs. Diese Hook-Funktion braucht deshalb nur dann implementiert werden, wenn das Standardverhalten nicht ausreicht.

Zur Textausgabe werden die bekannten curses-Funktionen wie "waddstr" o.ä. verwendet, wobei das zugehörige curses-Fenster über das CUIWINDOW-Datenelement "Frame" angesprochen wird!

3.4.7. SizeHook

Prototyp:

```
int SizeHook(void* win);
```

Der Size-Hook wird aufgerufen, wenn sich die Größe eines Fensters verändert hat. Falls die Darstellung des Fensters oder die Ausrichtung der Kindfenster von dieser Größenänderung betroffen sind, kann dies bei Bedarf innerhalb dieses Hooks angepasst werden.

Achtung: Die libCUI implementiert bereits ein Standardverhalten für die Größenänderung. Dabei werden alle maximierten Kindfenster auf die Größe des Client-Bereichs angepasst. Falls diese Funktionalität nicht erwünscht ist, sollte die Funktion "TRUE" zurückgeben und damit dem Fenster-Manager signalisieren, dass sie die Größenänderung vollständig selbst übernommen hat.

3.4.8. SetFocusHook

Prototyp:

```
void SetFocusHook(void* win, void* lastfocus);
```

Wird aufgerufen, wenn ein Fenster den Eingabefokus erhält. Der Parameter "lastfocus" enthält dabei eine Referenz auf das Fenster, das zuletzt im Besitz des Eingabefokus war.

Diese Funktion wird oftmals dazu verwendet, um die Sichtbarkeit des Cursors zu steuern.

3.4.9. KillFocusHook

Prototyp:

```
void KillFocusHook(void* win);
```

Wird aufgerufen, wenn einem Fenster der Eingabefokus entzogen wird.

Diese Funktion wird oftmals dazu verwendet, um die Sichtbarkeit des Cursors zu steuern.

3.4.10. KeyHook

Prototyp:

```
int KeyHook(void* win, int key);
```

Wird immer dann aufgerufen, wenn ein über die Tastatur eingegebenes Zeichen an das Fenster mit dem Eingabefokus gesendet werden soll. Das Fenster kann dann entspechend auf die Taste reagieren und mit dem Rückgabewert "TRUE" signalisieren, dass die Taste akzeptiert wurde. Wird statt dessen "FALSE" zurückgegeben, wird das Standardverhalten der libCUI aktiv.

Das Standardverhalten steuert die Weitergabe des Eingabefokus mit den Pfeiltasten oder der TAB- Taste, sowie die Standardbehandlung der ENTER- und der ESCAPE- Taste.

3.4.11. MMoveHook

Prototyp:

```
void MMoveHook(void* win, int x, int y);
```

Wird aufgerufen, wenn sich der Mauscursor über dem Fenster bewegt. Dies funktioniert nur dann, wenn die Mausbehandlung beim Aufruf von "WindowStart" aktiviert wurde und das Terminal diese Funktion unterstützt.

3.4.12. MButtonHook

Prototyp:

```
void MButtonHook(void* win, int x, int y, int flags);
```

Wird aufgerufen, wenn eine Maustaste über einem Fenster betätigt wurde. Der Parameter "flags" gibt dabei den von curses definierten Code für die jeweilige Aktion der Maustasten weiter.

Dies funktioniert nur dann, wenn die Mausbehandlung beim Aufruf von "WindowStart" aktiviert wurde und das Terminal diese Funktion unterstützt.

flags	Bedeutung
BUTTONx_PRESSED	Maustaste x gedrückt
BUTTONx_RELEASED	Maustaste x losgelassen
BUTTONx_CLICKED	Maustaste x kurzer Klick
BUTTONx_DOUBLE_CLICKED	Maustaste x Doppelklick
BUTTONx_TRIPLE_CLICKED	Maustaste x Dreifachklick

In obenstehender Tabelle ist x ein Platzhalter für Werte zwischen 1 und 4.

3.4.13. VScrollHook

Prototyp:

```
void VScrollHook(void* win, int sbcode, int pos);
```

Wird aufgerufen, wenn die vertikale Bildlaufleiste eines Fensters mit der Maus betätigt wurde. Dabei wird im Parameter ”sbcode” die Art der Betätigung weitergegeben:

sbcode	Bedeutung
SB_LINEDOWN	Zeilenweise nach unten scrollen
SB_LINEUP	Zeilenweise noch oben scrollen
SB_PAGEDOWN	Seitenweise nach unten scrollen
SB_PAGEUP	Seitenweise nach oben scrollen
SB_THUMBTRACK	Direktes Verschieben auf die Position ”pos”

Der Parameter ”pos” hat nur dann einen sinnvollen Wert, wenn ”sbcode” den Wert ”SB_-THUMBTRACK” enthält.

3.4.14. HScrollHook

Prototyp:

```
void HScrollHook(void* win, int sbcode, int pos);
```

Wird aufgerufen, wenn die horizontale Bildlaufleiste eines Fensters mit der Maus betätigt wurde. Dabei wird im Parameter ”sbcode” die Art der Betätigung weitergegeben:

sbcode	Bedeutung
SB_LINEDOWN	Spaltenweise nach rechts scrollen
SB_LINEUP	Spaltenweise noch links scrollen
SB_PAGEDOWN	Seitenweise nach rechts scrollen
SB_PAGEUP	Seitenweise nach links scrollen
SB_THUMBTRACK	Direktes Verschieben auf die Position ”pos”

Der Parameter ”pos” hat nur dann einen sinnvollen Wert, wenn ”sbcode” den Wert ”SB_-THUMBTRACK” enthält.

3.4.15. TimerHook

Prototyp:

```
void TimerHook(void* win, int id);
```

Wird aufgerufen, wenn ein für dieses Fenster aktivierter Window- Timer abgelaufen ist. Über den Parameter "id" kann dabei der jeweilige Timer identifiziert werden.

3.5. Darstellung und Bildschirm-Update

Wie die curses-Bibliothek, versucht auch die libCUI die Bildschirmausgaben so zu optimieren, dass möglichst wenige Daten an das Terminal gesendet werden müssen. Deshalb erscheint die Darstellung von Textausgaben in einem Fenster nicht unmittelbar am Bildschirm, sondern nur unter bestimmten Umständen:

Paint-Hook
Wird die Textausgabe innerhalb einer Paint- oder NcPaint- Ereignisbehandlung durchgeführt, dann wird anschließend der Bildschirminhalt automatisch als ungültig erklärt. Dies wiederum bewirkt automatisch die Aktualisierung des Terminals. Es handelt sich hierbei um den empfohlenen Standardfall. Soll aufgrund einer Statusänderung der Fensterinhalt verändert werden, kann der Aufruf der Paint-Routine über die Funktion "WindowInvalidate()" jederzeit erzwungen werden.

Direkte Textausgabe
Falls jedoch eine direkte Textausgabe in ein Fenster außerhalb der Paint-Routine erforderlich ist, muss anschließend der Bildschirminhalt manuell als ungültig erklärt werden. Dies geschieht mit der API-Funktion "WindowInvalidateScreen()". Der Fenster-Manager wird daraufhin die Aktualisierung des Bildschirminhaltes an geeigneter Stelle vornehmen.

Erzwungenes Bildschirmupdate
Muss zu einem definierten Zeitpunkt ein Bildschirmupdate unbedingt sein, dann kann dies mittels der Funktion "WindowUpdate()" ausgeführt werden. Allerdings ist zu beachten, dass aus Performance-Sicht erzwungene Bildschirmupdates die schlechteste Lösung sind.

3.6. Tastatureingabe und Eingabefokus

Tastatureingaben werden immer an dasjenige Fenster weitergeleitet, das gegenwärtig den Eingabefokus besitzt. Ist für dieses Fenster keine KeyHook- Funktion angemeldet oder gibt diese den Rückgabewert "FALSE" zurück, wird das Standardverhalten der libCUI aktiv.

Das Standardverhalten läuft dabei nach folgendem Schema ab:

Pfeil- und Tab-Tasten:
Der Eingabefokus wird an das nächste (bzw. je nach Taste an das vorherige) Kindfenster weitergeleitet, das sichtbar und aktiv ist und zudem den Fokus akzeptiert (CWS_TABSTOP).

Wird das Ende der Fensterliste der Kindfenster erreicht (oder existieren keine Kindfenster im aktuellen Fenster), dann wird die Aufgabe der Fokusweitergabe an das Vaterfenster delegiert. Dies geschieht jedoch nur, solange es sich beim aktuellen Fenster nicht um ein Popup-Fenster handelt.

Im Gegensatz zu normalen Fenstern beginnen Popup-Fenster beim Erreichen des Listenendes die Fokusweitergabe wieder am anderen Ende ihrer Fensterliste. Dadurch bildet die Weiterleitung des Fokus auf der Ebene des Popup-Fensters einen geschlossenen Kreis.

ENTER- und ESC- Tasten:
Es wird dasjenige Kindfenster gesucht, das den Fensterstil CWS_DEFOK (bzw. CWS_DEFCANCEL) besitzt. Wird ein solches Fenster gefunden und handelt es sich dabei um ein Button- Konrollelement, dann wird dieses betätigt.

Alphanumerische Tasten:
Es wird dasjenige Kindfenster gesucht, das das eingegebene Zeichen als Hot-Key hinterlegt hat. Falls ein solches Fenster gefunden wird, wird ihm der Eingabefokus zugewiesen und zugleich das fragliche Zeichen übergeben. Anderenfalls wird die Tastatureingabe verworfen.

Natürlich kann die Fokusweitergabe auch manuell im Programmcode beeinflusst werden. Es existieren dazu die folgenden Funktionen:

```
void WindowSetFocus(CUIWINDOW* win);
CUIWINDOW* WindowGetFocus(void);
void WindowFocusNext(CUIWINDOW* win);
void WindowFocusPrevious(CUIWINDOW* win);
```

3.7. Window-Timer

Soll ein sich zeitlich wiederholender Ablauf programmiert werden, dann kann dazu ein Window-Timer verwendet werden. Ein Window-Timer hat eine Auflösung von 100ms, besitzt aber keinen Anspruch auf Genauigkeit.

Um mit einem Timer zu Arbeiten, muss zunächst ein TimerHook für das entsprechende Fenster angemeldet werden. Dann wird der Timer mit der Funktion "WindowSetTimer" gestartet. Die Funktion "WindowKillTimer" deaktiviert den Timer wieder.

```
void WindowSetTimer(CUIWINDOW* win, int id, int msec);
void WindowKillTimer(CUIWINDOW* win, int id);
```

3.8. Farben

Wenn beim Funktionsaufruf "WindowStart" der Farbmodus aktiviert wurde und zudem die aktuelle Konsole Farben unterstützt, dann können während der Textausgabe in ein Fenster die Vorder- und die Hintergrundfarbe mit der Funktion "SetColor" gesetzt werden. Dazu existieren die folgenden Farbkonstanten (die den curses-Konstanten COLOR_XXXX entsprechen):

Konstante	Wert	Konstante	Wert
BLACK	0	DARKGRAY	8
BLACK	1	LIGHTRED	9
GREEN	2	LIGHTGREEN	10
BROWN	3	YELLOW	11
BLUE	4	LIGHTBLUE	12
MAGENTA	5	LIGHTMAGENTA	13
CYAN	6	LIGHTCYAN	14
LIGHTGRAY	7	WHITE	15

Die Funktion "SetColor" kennt vier Parameter. Dies sind das curses- Fenster auf das die Farbänderung angewendet werden soll, die Vordergrundfarbe und die Hintergrundfarbe. Der letzte Parameter gibt an, ob die Darstellung im monochromen Modus invers sein soll.

```
void SetColor(WINDOW* win, int fcolor, int bcolor, int reverse);
```

Allerdings kann es durchaus sinnvoll sein, die Farbkonstanten nicht direkt zu verwenden, sondern mit den Farben zu arbeiten, die dem Fenster hinterlegt sind. Dazu existiert innerhalb der CUIWINDOW- Struktur ein Datenelement "Color", das ebenfalls eine Struktur mit den folgenden Datenfeldern ist:

Datenfeld	Typ	Bedeutung
WndColor	int	normal window background color
WndSelColor	int	selected text background color
WndTxtColor	int	normal window text color
SelTxtColor	int	selected text color
InactTxtColor	int	inactive window text color
HilightColor	int	hilight window text color
TitleTxtColor	int	window caption text color
TitleBkgndColor	int	window caption background color
StatusTxtColor	int	window status bar text color
StatusBkgndColor	int	window status bar bkgnd color

Ein Code-Schnipsel aus einer Paint-Routine kann dann wie folgt aussehen:

```
if (index == data->SelIndex)
{
   SetColor(win->Win, win->Color.SelTxtColor,
            win->Color.WndSelColor, TRUE);
   cursor = y;
}
else
{
   if (win->IsEnabled)
   {
      SetColor(win->Win, win->Color.WndTxtColor,
               win->Color.WndColor, FALSE);
   }
   else
```

```
      {
         SetColor(win->Win, win->Color.InactTxtColor,
                  win->Color.WndColor, FALSE);
      }
   }
   for (x = 0; x < rc.W; x++)
   {
      if ((x > 0) && (x <= len))
      {
         wprintw(win->Win, "%c", item->ItemText[x - 1]);
      }
      else if (x == 0)
      {
         mvwprintw(win->Win, y, 0, " ");
      }
      else
      {
         wprintw(win->Win, " ");
      }
   }
```

Beim Anlegen einer Fensterstruktur erbt das neue Fenster seine Farben von seinem Vaterfenster. Es kann jedoch über die Funktion ”WindowColScheme” ein abweichendes Farbschema zugewiesen werden. Farbschemata haben Namen und werden über die Funktion ”WindowAddColScheme” hinzugefügt bzw. geändert. Standardmäßig sind die Schemata ”WINDOW”, ”DESKTOP”, ”DIALOG”, ”MENU”, ”TERMINAL” und ”HELP” definiert.

Die Funktionen im überblick:

```
void WindowAddColScheme(const char* name, CUIWINCOLOR* colrec);
int  WindowHasColScheme(const char* name);
void WindowColScheme(CUIWINDOW* win, const char* name);
```

Ein Beispiel:

```
CUIWINDOW* mynewwin;
CUIWINCOLOR* colrec;

colrec.WndColor = BLUE;
colrec.WndSelColor = LIGHTGRAY;
colrec.WndTxtColor = LIGHTGRAY;
colrec.SelTxtColor = BLACK;
colrec.InactTxtColor = DARKGRAY;
colrec.HilightColor = YELLOW;
colrec.TitleTxtColor = BLACK;
colrec.TitleBkgndColor = LIGHTGRAY;
colrec.StatusTxtColor = BLACK;
colrec.StatusBkgndColor = LIGHTGRAY;
WindowAddColScheme("MY_SCHEME", &colrec);
```

```
mynewwin = WindowNew(aparent, 0, 0, 10, 10, CWS_NONE);
WindowColScheme(mynewwin, "MY_SCHEME");
WindowCreate(mynewwin);
...
```

3.9. Bildlaufleisten

Ein Fenster kann eine vertikale und eine horizontale Bildlaufleiste (Scrollbalken) besitzen. Bildlaufleisten werden verwendet, um dem Anwender zu zeigen, an welcher Stelle innerhalb eines Bereiches (z.B. innerhalb einer Liste oder eines Textes) er sich befindet.

Ist ein Fenster mit der Eigenschaft CWS_BORDER versehen, dann finden die Bildlaufleisten auf dem Fensterrahmen Platz. Anderenfalls erfolgt die Darstellung der Scrollbalken am rechten bzw. unteren Fensterrand. Der Client-Bereich des Fensters wird dabei entsprechend verkleinert.

Mit den folgenden Funktionen werden Bildlaufleisten gesetzt und abgefragt:

```
void WindowEnableVScroll(CUIWINDOW* win, int enable);
void WindowEnableHScroll(CUIWINDOW* win, int enable);
void WindowSetVScrollRange(CUIWINDOW* win, int range);
void WindowSetHScrollRange(CUIWINDOW* win, int range);
void WindowSetVScrollPos(CUIWINDOW* win, int pos);
void WindowSetHScrollPos(CUIWINDOW* win, int pos);
int  WindowGetVScrollRange(CUIWINDOW* win);
int  WindowGetHScrollRange(CUIWINDOW* win);
int  WindowGetVScrollPos(CUIWINDOW* win);
int  WindowGetHScrollPos(CUIWINDOW* win);
```

Mittels ”WindowEnableVScroll” bzw. ”WindowEnableHScroll” wird die Sichtbarkeit gesteuert. Hat der Parameter ”enable” den Wert ”TRUE” dann wird die jeweilige Bildlaufleiste angezeigt, anderenfalls nicht.

Die Funktionen ”WindowSetVScrollRange” bzw. ”WindowSetHScrollRange” geben einer Bildlaufleiste den Scrollbereich vor, wobei der Wert des Parameters ”range” der Maximalwert ist, den die Position der Bildlaufleiste annehmen kann. Wird z.B. als Bereich der Wert ’1’ übergeben, dann kann die Position die Werte ’0’ und ’1’ annehmen.

Mit ”WindowSetVScrollPos” bzw. ”WindowSetHScrollPos” wird die Position innerhalb des Scrollbereichs gesetzt.

Mit den entsprechenden ”Get” Funktionen können die aktuellen Parameter der Bildlaufleiste abgefragt werden.

Ist die Mausbehandlung aktiv, dann können Bildlaufleisten auch mit der Maus bedient werden. Die Benachrichtgung an ein Fenster erfolgt dabei über den ”VScroll”- Hook bzw. Über den ”HScroll”- Hook des Fensters. Eine gängige Implementierung einer solchen Hook- Funktion sieht wie folgt aus:

```
static void
ListboxVScrollHook(void* w, int sbcode, int pos)
{
   CUIWINDOW* win = (CUIWINDOW*) w;
   CUIRECT rc;
   int sbpos, range;

   WindowGetClientRect(win, &rc);
   sbpos = WindowGetVScrollPos(win);
   range = WindowGetVScrollRange(win);

   switch(sbcode)
   {
   case SB_LINEUP:
      if (sbpos > 0)
      {
         WindowSetVScrollPos(win, sbpos - 1);
         WindowInvalidate(win);
      }
      break;
   case SB_LINEDOWN:
      if (sbpos < range)
      {
         WindowSetVScrollPos(win, sbpos + 1);
         WindowInvalidate(win);
      }
      break;
   case SB_PAGEUP:
      if (sbpos > 0)
      {
         sbpos -= (rc.H - 1);
         sbpos = (sbpos < 0) ? 0 : sbpos;
         WindowSetVScrollPos(win, sbpos);
         WindowInvalidate(win);
      }
      break;
   case SB_PAGEDOWN:
      if (sbpos < range)
      {
         sbpos += (rc.H - 1);
         sbpos = (sbpos > range) ? range : sbpos;
         WindowSetVScrollPos(win, sbpos);
         WindowInvalidate(win);
      }
      break;
   case SB_THUMBTRACK:
      WindowInvalidate(win);
```

```
            break;
        }
    }
```

3.10. Mausbehandlung

Die Behandlung der Maus wird in der libCUI über die Funktion ”WindowStart” aktiviert. Von da ab erhält ein Fenster, abhängig von den Möglichkeiten die das verwendete Terminal bietet, Informationen über die Mausbewegung und über Aktivitäten der Maustasten. Dazu dienen die beiden Hook- Funktionen ”MMoveHook” und ”MButtonHook”.

Die Mausereignisse werden immer an dasjenige Fenster gesendet, über dessen Client- Bereich sich der Maus-Cursor befindet. Allerdings kann ein Fenster die Maus ”einfangen” indem die Funktion ”WindowSetCapture” aufgerufen wird. Es werden von da an alle Ereignisse an das Fenster mit dem Maus- Capture gesendet, bis die Funktion ”WindowReleaseCapture” aufgerufen wird.

Wird eine Maustaste über einem Fenster betätigt, erhält dieses den Eingabefokus zugeteilt. Popup-Fenster werden zudem in den Vordergrund gestellt.

Die Behandlung von Mausereignissen im Randbereich (Nicht- Client Bereich) von Fenstern wird von der libCUI selbst verwaltet. Sollte das Standardverhalten nicht gewünscht sein, dann kann ein Fenster über die Hook-Funktionen ”MMoveHookNc” und ”MButtonHookNc” eine eigene Implementierung Anstelle der der libCUI einsetzen.

4. Allgemeine Fenterklassen

Wie im vorherigen Kapitel gezeigt wurde, kann mit der libCUI auf der Basis der allgemeinen Fensterstruktur CUIWINDOW eine Vielfalt unterschiedlicher Fensterklassen implementiert werden - alleine durch die Implementierung von Hook- Funktionen.

Neben dieser rudimentären Funktionalität bringt die LibCUI jedoch auch eine Reihe von Fensterklassen mit, die bereits fertig für ihre spezielle Aufgabe implementiert sind und vom Anwendungsprogrammierer ohne Änderung eingesetzt werden können. Dies sind vor allem Kontrollelemente wie z.B. Eingabezeilen, Buttons, Labels etc.

Dieses Kapitel stellt die einzelnen Kontrollelemente vor, ohne allzusehr auf deren Anwendung einzugehen. Hinweise zur Anwendung finden sich später im Kapitel "Dialoge".

4.1. Konventionen

Alle von der libCUI zur Verfügung gestellten Kontrollelemente befolgen ein paar gemeinsame Konventionen. Dies sind:

Erzeugung:
Ein libCUI Kontrollelement stellt eine Funktion zur Verfügung, mit der das Kontrollelement vollständig angelegt und spezialisiert wird. Üblicherweise trägt diese Funktion den Namen des Kontrollelements mit dem Anhängsel "New". Beispiel "ButtonNew". Das auf diese Weise angelegte Fenster braucht anschließend lediglich mittels der Funktion "WindowCreate" erzeugt zu werden.

Fensterstile:
Die Funktion zur Anlage von Kontrollelementinstanzen kennt zusätzlich zwei Parameter, die das Setzen von zusätzlichen Fensterstilen und das Zurücksetzen von standardmäßig gesetzen Fensterstilen erlauben. Wird hier jeweils CWS_NONE übergeben, dann wird das Kontrollelement mit den ihm eigenen Fensterstilen angelegt. Manche Kontrollelemente erweitern die Fensterstile der libCUI um spezifische Eigenschaften. So kennt z.B. die Eingabezeile den Stil "ED_PASSWORD".

Klassenname:
Jedes Kontrollelement hat seinen eigenen Klassennamen, den es an das Datenelement "Class" der Struktur CUIWINDOW gebunden hat. Der Klassenname entspricht dabei dem Namen des Kontrollelements. Beispiele: "EDIT", "BUTTON" und "LISTBOX"

Zugriffsfunktionen:
Ein Kontrollelement bringt eine Reihe von Funktionen mit, über die auf die jeweilige Ele-

mentinstanz und deren Daten zugegriffen werden kann. Diese Funktionen führen als Prefix immer den Namen des Kontrollelements. Beispiel: "EditSetText".

Custom Callback Hooks:
Während die im vorigen Kapitel vorgestellten Hook-Funktionen die Schnittstelle zwischen dem Fenstersystem und der Fensterinstanz bilden, bilden Custom Callback Hooks die Verbindung von Kontrollelementen zum Anwendungsprogramm. Über diese Hook- Funktionen teilt das Kontrollelement der Anwendung mit, wenn sich etwas geändert hat. Dies kann z.B. der Erhalt oder Verlust des Fokus sein, ebenso wie der Wechsel des markierten Eintrags in einer Listbox oder die Modifikation des Textes in einer Eingabezeile. Custom Callback Hooks müssen vom Anwendungsprogramm aus beim jeweiligen Kontrollelement angemeldet werden.

Kennnummer (ID):
Jedes Kontrollelement erhält eine (innerhalb des aktuellen Popup-Fensters) eindeutige Kennnummer. Über diese ID kann das Kontrollelement jederzeit mittels der Funktion "WindowGetCtrl" gefunden werden.

Hot-Key:
Kontrollelemente kennen einen Hot-Key und können darüber direkt durch einen Tastendruck bedient werden, auch wenn der Eingabefokus auf einem anderen Fenster steht. Dies funktioniert natürlich nur dann, wenn das andere Fenster die eingegebene Taste nicht sebst verwendet. Als Hot-Key wird dasjenige Zeichen hinterlegt, das im Fenstertext mit einem vorangestellten '&' markiert ist.

4.2. Label

Ein Label ist ein statisches Textfeld, in dem einzeiliger oder mehrzeiliger Text dargestellt werden kann. Es wird häufig für Beschiftungen innerhalb von Dialogen verwendet.

Anlage:

```
CUIWINDOW* LabelNew(CUIWINDOW* parent,
                    const wchar_t* text,
                    int x, int y, int w, int h,
                    int id,
                    int sflags, int cflags);
```

Standardstil:

```
CWS_NONE
```

Custom Callbacks:

```
void LabelSetSetFocusHook (CUIWINDOW* win,
                           CustomHook1PtrProc proc,
                           CUIWINDOW* target);
void LabelSetKillFocusHook(CUIWINDOW* win,
                           CustomHookProc proc,
                           CUIWINDOW* target);
```

Beispiel:

```
ctrl = LabelNew(aparent, "Password:", 2, 1, 10, 1,
                IDC_STATIC, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);
```

4.3. Button

Ein Button ist eine Schaltfläche, die mit der Enter- oder der Leertaste betätigt (geschaltet) werden kann. Buttons werden häufig zum Schließen von Fenstern o.ä. verwendet.

Anlage:

```
CUIWINDOW* ButtonNew(CUIWINDOW* parent,
                     const wchar_t* text,
                     int x, int y, int w, int h,
                     int id,
                     int sflags, int cflags)
```

Standardstil:

```
CWS_TABSTOP
```

Erweiterte Stile:

```
CWS_DEFOK, CWS_DEFCANCEL
```

Custom Callbacks:

```
void ButtonSetSetFocusHook  (CUIWINDOW* win,
                             CustomHook1PtrProc proc,
                             CUIWINDOW* target);
void ButtonSetKillFocusHook (CUIWINDOW* win,
                             CustomHookProc proc,
                             CUIWINDOW* target);
void ButtonSetClickedHook   (CUIWINDOW* win,
                             CustomHookProc proc,
                             CUIWINDOW* target);
```

Beispiel:

```
ctrl = ButtonNew(aparent, "&Ok", 15, 6, 10, 1,
                 IDOK, CWS_DEFOK, CWS_NONE);
ButtonSetClickedHook(ctrl, MyOkHook, aparent);
WindowCreate(ctrl);
...

void MyOkHook(void* w, void* c)
{
   WindowClose((CUIWINDOW*) w, IDOK);
}
```

4.4. Checkbox

Mit Hilfe von Checkboxen können Werte dargestellt und verändert werden, die nur die zwei Zustände "markiert" und "nicht markiert" kennen. Befinden sich mehrere Checkboxen in einem gemeinsamen Fensterbereich (z.B. in einer Groupbox), dann können ihre Werte unabhängig voneinander verändert werden.

Anlage:

```
CUIWINDOW* CheckboxNew(CUIWINDOW* parent,
                       const wchar_t* text,
                       int x, int y, int w, int h,
                       int id,
                       int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP
```

Custom Callbacks:

```
void CheckboxSetSetFocusHook (CUIWINDOW* win,
                              CustomHook1PtrProc proc,
                              CUIWINDOW* target);
void CheckboxSetKillFocusHook(CUIWINDOW* win,
                              CustomHookProc proc,
                              CUIWINDOW* target);
void CheckboxSetClickedHook  (CUIWINDOW* win,
                              CustomHookProc proc,
                              CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void CheckboxSetCheck(CUIWINDOW* win, int state);
int  CheckboxGetCheck(CUIWINDOW* win);
```

Beispiel:

```
ctrl = CheckboxNew(agroup, "Create &Databases", 1, 2, 20, 1,
                   IDC_CHKCREATEDB, CWS_NONE, CWS_NONE);
CheckboxSetCheck(ctrl, data->CreateDb);
WindowCreate(ctrl);

ctrl = CheckboxNew(agroup, "Create &Users", 1, 3, 20, 1,
                   IDC_CHKCREATEUSER, CWS_NONE, CWS_NONE);
CheckboxSetCheck(ctrl, data->CreateUser);
WindowCreate(ctrl);

...

ctrl = WindowGetCtrl(win, IDC_CHKCREATEDB);
```

```
if (ctrl)
{
   data->CreateDb = CheckboxGetCheck(ctrl);
}

...
```

4.5. Radio-Button

Wie die Checkbox, so ist auch der Radio-Button ein Kontrollelement mit dem Werte dargestellt und verändert werden können, die nur die zwei Zustände ”markiert” und ”nicht markiert” besitzen. Im Unterschied zur Checkbox kann in einem gemeinsamen Fensterbereich (z.B. in einer Groupbox) jedoch immer nur einer der Radio-Buttons eine Markierung haben, die übrigen werden automatisch auf ”nicht markiert” gesetzt.

Anlage:

```
CUIWINDOW* RadioNew(CUIWINDOW* parent,
                    const wchar_t* text,
                    int x, int y, int w, int h,
                    int id,
                    int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP
```

Custom Callbacks:

```
void RadioSetSetFocusHook (CUIWINDOW* win,
                           CustomHook1PtrProc proc,
                           CUIWINDOW* target);
void RadioSetKillFocusHook(CUIWINDOW* win,
                           CustomHookProc proc,
                           CUIWINDOW* target);
void RadioSetClickedHook  (CUIWINDOW* win,
                           CustomHookProc proc,
                           CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void RadioSetCheck(CUIWINDOW* win, int state);
int  RadioGetCheck(CUIWINDOW* win);
```

Beispiel:

```
ctrl = RadioNew(agroup, "&1 kHz sample rate", 1, 2, 20, 1,
                  IDC_RB1KHZ, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);
```

```
ctrl = RadioNew(agroup, "&4 kHz sample rate", 1, 3, 20, 1,
                    IDC_RB4KHZ, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

ctrl = RadioNew(agroup, "&8 kHz sample rate", 1, 4, 20, 1,
                    IDC_RB8KHZ, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

ctrl = NULL;
switch(data->SampleRate)
{
case 1: ctrl = WindowGetCtrl(agroup, IDC_RB1KHZ); break;
case 4: ctrl = WindowGetCtrl(agroup, IDC_RB4KHZ); break;
case 8: ctrl = WindowGetCtrl(agroup, IDC_RB8KHZ); break;
default: ctrl = WindowGetCtrl(agroup, IDC_RB1KHZ); break;
}

if (ctrl)
{
   RadioSetCheck(ctrl, TRUE);
}
```

4.6. Edit

Ein Edit-Kontrollelement wird zur einzeiligen Darstellung und Eingabe von Textdaten herangezogen. Optional kann es in einem Passwort-Modus betrieben werden, in dem anstelle der eingegebenen Zeichen der Platzhalter '*' dargestellt wird.

Anlage:

```
CUIWINDOW* EditNew (CUIWINDOW* parent,
                    const wchar_t* text,
                    int x, int y, int w, int h,
                    int len,
                    int id,
                    int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP
```

Erweiterte Stile:

```
EF_PASSWORD
```

Custom Callbacks:

```
void EditSetSetFocusHook (CUIWINDOW* win,
                          CustomHook1PtrProc proc,
```

```
                            CUIWINDOW* target);
void EditSetKillFocusHook(CUIWINDOW* win,
                            CustomHookProc proc,
                            CUIWINDOW* target);
void EditSetChangedHook  (CUIWINDOW* win,
                            CustomHookProc proc,
                            CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void EditSetText(CUIWINDOW* win, const wchar_t* text);
const wchar_t* EditGetText(CUIWINDOW* win, wchar_t* text, int len);
```

Beispiel:

```
ctrl = EditNew(aparent, "", 64, 1, 2, 20, 1,
               IDC_EDPASSWORD, EF_PASSWORD, CWS_NONE);
WindowCreate(ctrl);
EditSetText(ctrl, "sesam123");
```

4.7. Listbox

Die Listbox stellt eine einspaltige scrollbare Auswahlliste zur Verfügung, in der ein Eintrag markiert werden kann. Listboxen können alle möglichen Daten enthalten und dem Anwender zur Auswahl anbieten.

Anlage:

```
CUIWINDOW* ListboxNew(CUIWINDOW* parent,
                      const wchar_t* text,
                      int x, int y, int w, int h,
                      int id,
                      int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP, CWS_BORDER
```

Erweiterte Stile:

```
LB_SORTED, LB_DESCENDING
```

Custom Callbacks:

```
void ListboxSetSetFocusHook  (CUIWINDOW* win,
                              CustomHook1PtrProc proc,
                              CUIWINDOW* target);
void ListboxSetKillFocusHook (CUIWINDOW* win,
                              CustomHookProc proc,
                              CUIWINDOW* target);
void ListboxSetLbChangedHook (CUIWINDOW* win,
```

```
                                CustomHookProc proc,
                                CUIWINDOW* target);
void ListboxSetLbChangingHook(CUIWINDOW* win,
                                CustomBoolHookProc proc,
                                CUIWINDOW* target);
void ListboxSetLbClickedHook (CUIWINDOW* win,
                                CustomHookProc proc,
                                CUIWINDOW* target);
```

Zugriffsfunktionen:

```
int  ListboxAdd(CUIWINDOW* win, const wchar_t* text);
void ListboxDelete(CUIWINDOW* win, int index);
const wchar_t* ListboxGet(CUIWINDOW* win, int index);
void ListboxSetData(CUIWINDOW* win, int index,
                    unsigned long data);
unsigned long ListboxGetData(CUIWINDOW* win, int index);
void ListboxSetSel(CUIWINDOW* win, int index);
int  ListboxGetSel(CUIWINDOW* win);
void ListboxClear(CUIWINDOW* win);
int  ListboxGetCount(CUIWINDOW* win);
```

Beispiel:

```
ctrl = ListboxNew(aparent, "Names", 1, 2, 20, 10,
                IDC_LBNAMES, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

ListboxAdd(ctrl, "Otto");
ListboxAdd(ctrl, "Fritz");
ListboxAdd(ctrl, "Maria");
ListboxAdd(ctrl, "Franz");

ListboxSetSel(ctrl, 0);
```

4.8. Listview

Der Listview ist der Listbox sehr verwandt, mit dem Unterschied, dass hier die Daten in mehreren Spalten dargestellt werden können. Die Anzahl der Spalten wird bei der Anlage festgelegt, die Angabe der Spaltenüberschrifen erfolgt später.

Ein Eintrag in der Liste wird über die folgende Struktur abgebildet:

```
typedef struct LISTRECStruct
{
   wchar_t**        ColumnText;   /* char array with text data */
   int*          ColumnWidth;  /* Width of column */
   int           NumColumns;   /* Numer of columns */
```

```
    unsigned long Data;         /* User data */
    void*         Next;         /* Next list record */
 } LISTREC;
```

Anlage:

```
CUIWINDOW* ListviewNew(CUIWINDOW* parent,
                       const wchar_t* text,
                       int x, int y, int w, int h,
                       int num_cols,
                       int id,
                       int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP, CWS_BORDER
```

Custom Callbacks:

```
void ListviewSetSetFocusHook  (CUIWINDOW* win,
                               CustomHook1PtrProc proc,
                               CUIWINDOW* target);
void ListviewSetKillFocusHook (CUIWINDOW* win,
                               CustomHookProc proc,
                               CUIWINDOW* target);
void ListviewSetLbChangedHook (CUIWINDOW* win,
                               CustomHookProc proc,
                               CUIWINDOW* target);
void ListviewSetLbChangingHook(CUIWINDOW* win,
                               CustomBoolHookProc proc,
                               CUIWINDOW* target);
void ListviewSetLbClickedHook (CUIWINDOW* win,
                               CustomHookProc proc,
                               CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void ListviewAddColumn(CUIWINDOW* win,
                       int colnr,
                       const wchar_t* text);

void ListviewClear(CUIWINDOW* win);

LISTREC* ListviewCreateRecord(CUIWINDOW* win);

int  ListviewInsertRecord(CUIWINDOW* win,
                          LISTREC* newrec,
                          int doupdate);

void ListviewSetColumnText(LISTREC* rec,
                           int colnr,
```

```
                              const wchar_t* text);

const wchar_t* ListviewGetColumnText(LISTREC* rec,
                                     int colnr);

void ListviewSetSel(CUIWINDOW* win, int index);
int  ListviewGetSel(CUIWINDOW* win);
LISTREC* ListviewGetRecord(CUIWINDOW* win, int index);
int  ListviewGetCount(CUIWINDOW* win);
```

Beispiel:

```
ctrl = ListviewNew(aparent, "Address",
                   1, 2, 40, 10, 3,
                   IDC_LVADDR, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

ListviewAddColumn(ctrl, 0, "Name");
ListviewAddColumn(ctrl, 1, "Firstname");
ListviewAddColumn(ctrl, 2, "Town");

rec = ListviewCreateRecord(ctrl);
if (rec)
{
   ListviewSetColumnText(rec, 0, "Mustermann");
   ListviewSetColumnText(rec, 1, "Max");
   ListviewSetColumnText(rec, 2, "Hintertupfingen");
   ListviewInsertRecord(ctrl, rec, FALSE);
}
rec = ListviewCreateRecord(ctrl);
if (rec)
{
   ListviewSetColumnText(rec, 0, "R"ussel");
   ListviewSetColumnText(rec, 1, "Rudi");
   ListviewSetColumnText(rec, 2, "Musterhausen");
   ListviewInsertRecord(ctrl, rec, TRUE);
}

ListviewSetSel(ctrl, 0);
```

4.9. Textview

Der Textview erlaubt das Betrachten von vielzeiligem Text in einer scrollbaren Ansicht. Optional kann der Text am Zeilenende umgebrochen werden. Texte werden entweder über Zugriffsfunktionen in das Kontrollelement übertragen oder aus einer Datei gelesen.

Anlage:

```
CUIWINDOW* TextviewNew(CUIWINDOW* parent,
                       const wchar_t* text,
                       int x, int y, int w, int h,
                       int id,
                       int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP, CWS_BORDER
```

Custom Callbacks:

```
void TextviewSetSetFocusHook  (CUIWINDOW* win,
                               CustomHook1PtrProc proc,
                               CUIWINDOW* target);
void TextviewSetKillFocusHook (CUIWINDOW* win,
                               CustomHookProc proc,
                               CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void TextviewEnableWordWrap(CUIWINDOW* win, int enable);
void TextviewAdd(CUIWINDOW* win, const wchar_t* text,
                 int doupdate);
void TextviewClear(CUIWINDOW* win);
int  TextviewRead(CUIWINDOW* win, const wchar_t* filename);
int  TextviewSearch(CUIWINDOW* win,
                 const wchar_t* text,
                 int wholeword,
                 int casesens,
                 int down);
```

Beispiel:

```
ctrl = TextviewNew(aparent, "User Accounts",
                   1, 2, 40, 10,
                   IDC_TVACCOUNTS, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

if (!TextviewRead(ctrl, "/etc/passwd"))
{
   MessageBox(aparent,
              "Error reading file /etc/passwd!",
              "Error",
              MB_ERROR);
}
```

4.10. Progressbar

Der PogressBar ist ein Kontrollelement mit dem der Fortschritt eines Vorgangs visualisiert werden kann, der längere Zeit in Anspruch nimmt. Vergleichbar mit einer Bildlaufleiste kennt der Fortschrittsbalken einen Bereich und eine Position.

Anlage:

```
CUIWINDOW* ProgressbarNew(CUIWINDOW* parent,
                          const wchar_t* text,
                          int x, int y, int w, int h,
                          int id,
                          int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP, CWS_BORDER
```

Zugriffsfunktionen:

```
void ProgressbarSetRange(CUIWINDOW* win, int range);
void ProgressbarSetPos(CUIWINDOW* win, int pos);
int  ProgressbarGetRange(CUIWINDOW* win);
int  ProgressbarGetPos(CUIWINDOW* win);
```

Beispiel:

```
ctrl = ProgressbarNew(aparent, "Progress",
                   1, 2, 30, 3,
                   IDC_PGBAR, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

ProgressbarSetRange(ctrl, 100);
ProgressbarSetPos(ctrl, 50);
```

4.11. Memo

Das Memo-Kontrollelement erlaubt die Bearbeitung von mehrzeiligem Text in einem Editor-Fenster. Eine Textzeile kann dabei eine definierbare maximale Breite annehmen, ab der das folgende Wort in die nächste Zeile umgebrochen wird (word wrap). Der Umbruch erfolgt an Leerzeichen und Bindestrichen.

Alternativ kann das Kontrollelement in einem automatischen Umbruchsmodus betrieben werden, in dem der Text immer am rechten Fensterrand umgebrochen wird und dadurch niemals breiter als der tatsächlich sichtbare Bereich des Fensterinhaltes ist. Wird das Kontrollelement in seiner Grösse verändert, wird der Text automatisch auf die neue Fensterbreite umgerechnet.

Anlage:

```
CUIWINDOW* MemoNew (CUIWINDOW* parent,
                    const wchar_t* text,
                    int x, int y, int w, int h,
                    int id,
                    int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP
```

Erweiterte Stile:

```
MF_AUTOWORDWRAP
```

Custom Callbacks:

```
void MemoSetSetFocusHook (CUIWINDOW* win,
                          CustomHook1PtrProc proc,
                          CUIWINDOW* target);
void MemoSetKillFocusHook(CUIWINDOW* win,
                          CustomHookProc proc,
                          CUIWINDOW* target);
void MemoSetChangedHook  (CUIWINDOW* win,
                          CustomHookProc proc,
                          CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void        MemoSetText(CUIWINDOW* win, const wchar_t* text);
const wchar_t* MemoGetText(CUIWINDOW* win, wchar_t* text, int len);
int         MemoGetTextBufSize(CUIWINDOW* win);
void        MemoSetWrapColumns(CUIWINDOW* win, int cols);
```

Beispiel:

```
ctrl = MemoNew(aparent, "", 1, 2, 20, 10,
               IDC_MEMO, MF_AUTOWORDWRAP, CWS_NONE);
WindowCreate(ctrl);
MemoSetText(ctrl, "Hello World this is a test");
```

4.12. Terminal

Das Terminal Kontrollelement erlaubt es, Co- Prozesse in einem interaktiven Fenster ablaufen zu lassen. Der Co- Prozess wird im Hintergrund gestartet und über Pipes mit dem Kontrollelements verbunden.

Anlage:

```
CUIWINDOW* TerminalNew(CUIWINDOW* parent,
                       const wchar_t* text,
                       int x, int y, int w, int h,
```

```
                          int id,
                          int sflags, int cflags);
```

Standardstil:

```
CWS_TABSTOP, CWS_BORDER
```

Custom Callbacks:

```
void TerminalSetSetFocusHook (CUIWINDOW* win,
                              CustomHook1PtrProc proc,
                              CUIWINDOW* target);
void TerminalSetKillFocusHook(CUIWINDOW* win,
                              CustomHookProc proc,
                              CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void TerminalWrite(CUIWINDOW* win, const wchar_t* text,
                   int numchar, int doupdate);
int  TerminalRun(CUIWINDOW* win, const wchar_t* cmd);
```

Beispiel:

```
ctrl = TerminalNew(aparent, "Shell",
                   1, 2, 40, 10,
                   IDC_TERMINAL, CWS_NONE, CWS_NONE);
WindowCreate(ctrl);

TerminalWrite(ctrl, "Just type 'exit' to return\r\n");
TerminalRun(ctrl, "/bin/bash -i");
```

4.13. Menu

Ein Menü ist ein Kontrollelement, das dem Anwender erlaubt aus einer Liste von Optionen (Menübefehlen) zu wählen. Jeder Menüeintrag besitzt dabei eine Nummer und einen Namen. Ein als "moveable" markierter Menüeintrag kann im Menü verschoben werden, wenn das Kontrollelement in den "Drag" Mode geschaltet wird.

Die Auswahl eines Menüeintrags erfolgt entweder mittels der Pfeiltasten, über die direkte Eingabe der Nummer oder mit Hilfe der Maus.

Ein Menü kann in einer Liste von Kindfenstern sowie als Popup Fenster eingesetzt werden.

Menüpunkte werden in einer Struktur verwaltet, die den folgenden Aufbau hat:

```
typedef struct MENUITEMStruct
{
   wchar_t*         ItemText;
   int           IsSeparator;
   int           IsMoveable;
```

```
    unsigned long ItemId;
    void*         Next;
    void*         Previous;
} MENUITEM;
```

Anlage:

```
CUIWINDOW* MenuNew(CUIWINDOW* parent,
                   const wchar_t* text,
                   int x, int y, int w, int h,
                   int id,
                   int sflags, int cflags);
```

Custom Callbacks:

```
void MenuSetSetFocusHook    (CUIWINDOW* win,
                             CustomHook1PtrProc proc,
                             CUIWINDOW* target);
void MenuSetKillFocusHook   (CUIWINDOW* win,
                             CustomHookProc proc,
                             CUIWINDOW* target);
void MenuSetMenuChangedHook (CUIWINDOW* win,
                             CustomHookProc proc,
                             CUIWINDOW* target);
void MenuSetMenuChangingHook(CUIWINDOW* win,
                             CustomBoolHookProc proc,
                             CUIWINDOW* target);
void MenuSetMenuClickedHook (CUIWINDOW* win,
                             CustomHookProc proc,
                             CUIWINDOW* target);
void MenuSetMenuEscapeHook  (CUIWINDOW* win,
                             CustomHookProc proc,
                             CUIWINDOW* target);
```

Zugriffsfunktionen:

```
void MenuAddItem(CUIWINDOW* win,
                 const wchar_t* text,
                 unsigned long id,
                 int moveable);
void MenuAddSeparator(CUIWINDOW* win, int moveable);
void MenuSelectItem(CUIWINDOW* win, unsigned long id);
MENUITEM* MenuGetSelectedItem(CUIWINDOW* win);
void MenuSetDragMode(CUIWINDOW* win, int state);
void MenuClear(CUIWINDOW* win);
```

Beispiel:

```
CUIWINDOW* menu = MenuNew(win, "Edit", 0, 0, 30, 12, 1,
                          CWS_CENTERED | CWS_POPUP, CWS_NONE);
if (menu)
```

```
{
   int choice = 0;

   MenuAddItem(menu, "Edit selected database", 1, FALSE);
   MenuAddItem(menu, "Drop selected database", 2, FALSE);
   MenuAddItem(menu, "Create new database", 3, FALSE);
   MenuAddSeparator(menu, FALSE);
   MenuAddItem(menu, "Exit application", 7, FALSE);
   MenuAddSeparator(menu, FALSE);
   MenuAddItem(menu, "Close menu", 0, FALSE);
   MenuSelectItem(menu, 1);

   MenuSetMenuClickedHook(menu, DatabasesMenuChoice, win);
   MenuSetMenuEscapeHook(menu, DatabasesMenuEscape, win);

   WindowCreate(menu);
   if (WindowModal(menu) == IDOK)
   {
      MENUITEM* item = MenuGetSelectedItem(menu);
      if (item)
      {
         choice = item->ItemId;
      }
      WindowDestroy(menu);

      switch(choice)
      {
      case 1:
         DatabasesModify(win, &data->Module[data->Current]);
         break;
      case 2:
         DatabasesDelete(win, &data->Module[data->Current]);
         break;
      case 3:
         DatabasesCreate(win, &data->Module[data->Current]);
         break;
      case 7:
         WindowQuit(EXIT_SUCCESS);
         break;
      }
   }
   else
   {
      WindowDestroy(menu);
   }
}
```

5. Dialoge

Dialoge sind Fenster, die als Kindfenster Kontrollelemente enthalten, über die sie mit dem Anwender kommunizieren. Das Dialogfenster stellt dabei eine Infrastruktur zur Verfügung, die die korrekte Funktion der Kontrollelemente im Kontext des Dialogs ermöglicht.

In der libCUI gibt es aus technischer Sicht keinen Unterschied zwischen Fenstern und Dialogen, da das Standardverhalten eines libCUI-Fenster das eines Dialogs ist. Der folgende Abschnitt geht auf ein paar Besonderheiten ein, die es trotzdem zu beachten gilt.

5.1. Modale und nichtmodale Dialoge

Ein Dialog sollte grundsätzlich ein Popup-Fenster sein. Für Popup- Fenster gilt dabei generell, dass es für sie zwei Ausführungsmodi gibt: Modal und nichtmodal.

Ein nichtmodales Popup-Fenster überlagert das Anwendungsfenster und wird immer über diesem angeordnet. Allerdings kann nach wie vor auch das Hauptfenster (bzw. seine Kindfenster) Windows-Nachrichten empfangen und den Eingabefokus erhalten. Das Popupfenster unterbindet also nicht den Zugriff auf den Teil der Anwendung der darunter liegt. Diese Form der Popup-Fenster macht nur in den seltensten Fällen wirklich Sinn.

Ein modales Popup-Fenster liegt ebenfalls immer über dem Anwendungsfenster, sperrt aber zudem den Zugriff darauf, solange bis das Popup-Fenster geschlossen wird. Dialogfenster sind in der Regel immer modal.

Beispiel für ein nichtmodales Popup-Fenster:

```
dlg = MyWindowNew(win, 10, 2, 40, 10, CWS_POPUP, CWS_NONE);
WindowCreate(dlg);
WindowSetFocus(dlg);
...
```

Beispiel für ein modales Popup-Fenster:

```
dlg = LoginDlgNew(win, show_server, CWS_NONE, CWS_NONE);
if (dlg)
{
   WindowCreate(dlg);
   if (WindowModal(dlg) == IDOK)
   {
      /* process dialog data */
   }
   WindowDestroy(dlg);
```

```
}
```

Der entscheidende Unterschied im zweiten Beispiel ist der API Aufruf ”WindowModal” über den das Fenster ”dlg” in einer modalen Schleife ausgeführt wird. Der Programmablauf wird scheinbar beim Funktionsaufruf von ”WindowModal” angehalten bis der Dialog vom Anwender geschlossen wurde. Der Rückgabewert der Funktion ist der Result-Code, der der Funktion ”WindowClose” innerhalb des Dialogs übergeben wurde.

Obwohl es für den Rückgabewert eines Dialogs keine funktionale Festlegung gibt, sollte für Standardfälle eine der von der libCUI definierten Konstanten verwendet werden:

```
#define IDOK              0x00000001
#define IDCANCEL          0x00000002
#define IDYES             0x00000003
#define IDNO              0x00000004
#define IDRETRY           0x00000005
```

Im Folgenden wird immer von modalen Dialogen ausgegangen, wenn von Dialogen die Rede ist.

5.2. Dialogdaten

Dialoge sind in der Regel so aufgebaut, dass ihre Datenfelder vom aufrufenden Programmteil initialisiert werden und nach dem Schließen des Dialogs dort auch wieder ausgewertet werden. Eine mögliche Vorgehensweise dazu ist, die Instanzdaten des Dialogfensters von über einen Zeiger sichtbar zu machen. Oder die Daten über eine Austauschstruktur in den Dialog zu kopieren. Auch der Zugriff auf einzelne Datenfelder über separate Zugriffsfunktionen ist denkbar, jedoch recht aufwändig. Die gewählte Vorgehensweise ist Geschmacksache und bleibt dem Programmierer überlassen.

Hier als Beipiel ein Auszug aus der Header-Datei eines Login-Dialogs:

```
typedef struct
{
   char Host[128 + 1];
   char Port[32 + 1];
   char Username[64 + 1];
   char Password[64 + 1];
   int  ShowServer;
} LOGINDLGDATA;

CUIWINDOW* LoginDlgNew(CUIWINDOW* parent,
                       int show_server,
                       int sflags, int cflags);
LOGINDLGDATA* LoginDlgGetData(CUIWINDOW* win);
```

In der Funktion ”LoginDlgNew” wird dabei eine Instanz des Datentyps ”LOGINDLGDATA” angelegt und dem ”InstData” Zeiger der Fensterstruktur zugewiesen. Die Funktion ”LoginDlgGetData” erlaubt den typsicheren Zugriff auf das Datenelement.

```
    dlg = LoginDlgNew(win, show_server, CWS_NONE, CWS_NONE);
    if (dlg)
    {
       LOGINDLGDATA* dlgdata = LoginDlgGetData(dlg);
       if (dlgdata)
       {
          strcpy(dlgdata->Username, "postgres");
          strcpy(dlgdata->Password, "");

          WindowCreate(dlg);
          if (WindowModal(dlg) == IDOK)
          {
             Login(dlgdata->Username, dlgdata->Password);
          }
       }
       WindowDestroy(dlg);
    }
```

Hingewiesen werden sollte noch auf die Tatsache, dass der Aufruf von "WindowCreate" erst erfolgt, nachdem die Datenfelder initialisiert wurden, da hier im CreateHook die Kontrollelemente angelegt und mit den Werten aus den Dialogdaten gefüttert werden. Sollte nach dem Erzeugen des Dialogs ein Datentransfer in die Kontrollelemente stattfinden, dann muss der Dialog eine geeignete Update-Funktion bieten.

5.3. Dialogfenster anlegen

Wie bereits bei den Kontrollelementen gezeigt, können alle zur Anlage und zur Spezialisierung erforderlichen Schritte auch bei Dialogen in einer gemeinsamen Funktion zusammengelegt werden, so dass das resultierende Fenster nurnoch erzeugt werden muss, bevor es an "WindowModal" übergeben werden kann. Welche Schritte dabei i.d.R. unternommen werden, zeigt das folgende Beispiel:

```
CUIWINDOW*
LoginDlgNew(CUIWINDOW* parent, int show_server,
            int sflags, int cflags)
{
   if (parent)
   {
      CUIWINDOW* dlg;
      int flags = sflags | CWS_POPUP | CWS_BORDER | CWS_CENTERED;
      flags &= ~(cflags);

      if (!show_server)
      {
         dlg = WindowNew(parent, 0, 0, 40, 9, flags);
      }
```

```
      else
      {
         dlg = WindowNew(parent, 0, 0, 40, 13, flags);
      }
      dlg->Class = "LOGIN_DLG";

      WindowColScheme(dlg, "DIALOG");
      WindowSetCreateHook(dlg, LoginDlgCreateHook);
      WindowSetDestroyHook(dlg, LoginDlgDestroyHook);

      dlg->InstData = (LOGINDLGDATA*)
          malloc(sizeof(LOGINDLGDATA));
      ((LOGINDLGDATA*)dlg->InstData)->Username[0] = 0;
      ((LOGINDLGDATA*)dlg->InstData)->Password[0] = 0;
      ((LOGINDLGDATA*)dlg->InstData)->Host[0] = 0;
      ((LOGINDLGDATA*)dlg->InstData)->Port[0] = 0;
      ((LOGINDLGDATA*)dlg->InstData)->ShowServer =
          show_server;

      WindowSetText(dlg, "Login");
      return dlg;
   }
   return NULL;
}
```

Zunächst wird ein normales libCUI Popup-Fenster angelegt, das es, abhängig vom Parameter 'show_server' in zwei unterschiedlichen Varianten geben kann. Zudem wird dem neuen Fenster der Klassenname "LOGIN_DLG" zugewiesen.

Anschließend wird das Farbschema für Dialoge zugeordnet und zwei Hook- Funktionen angemeldet - auf die später noch eingegangen wird. Dann erfolgt die Instanzierung und Initialisierung des Datentyps "LOGINDLGDATA" sowie die Zuweisung an den Instanzdatenzeiger des Dialogfensters.

Zuletzt wird dem Dialogfenster der Fenstertext "Login" zugewiesen, der in der Titlezeile des Fensters angezeigt werden wird.

5.4. Kontrollelemente verwalten

Wie aus dem vorigen Abschnitt ersichtlich wird, werden die Kontrollelemente nicht schon bei der Anlage des Dialogs erzeugt. Vielmehr wird dazu der CreateHook verwendet, der im obigen Beispiel unter dem Namen "LoginDlgCreateHook" rangiert. Eine Implementierung kann dabei wie folgt aussehen:

```
static void
LoginDlgCreateHook(void* w)
{
```

```
    CUIWINDOW* win = (CUIWINDOW*) w;
    CUIWINDOW* ctrl;
    LOGINDLGDATA* data = (LOGINDLGDATA*) win->InstData;

    ctrl = LabelNew(win, "User name:",
                    2, 1, 12, 1, 0,
                    CWS_NONE, CWS_NONE);
    WindowCreate(ctrl);

    ctrl = LabelNew(win, "Password:",
                    2, 3, 12, 1, 0,
                    CWS_NONE, CWS_NONE);
    WindowCreate(ctrl);

    ctrl = EditNew(win, data->Username,
                    14, 1, 20, 1, 64,
                    IDC_EDUSERNAME,
                    CWS_NONE, CWS_NONE);
    WindowCreate(ctrl);

    ctrl = EditNew(win, data->Password,
                    14, 3, 20, 1, 64,
                    IDC_EDPASSWD,
                    EF_PASSWORD, CWS_NONE);
    WindowSetFocus(ctrl);
    WindowCreate(ctrl);

    ctrl = ButtonNew(win, "&OK",
                    7, 5, 10, 1,
                    IDOK,
                    CWS_DEFOK, CWS_NONE);
    ButtonSetClickedHook(ctrl, LoginDlgButtonHook, win);
    WindowCreate(ctrl);

    ctrl = ButtonNew(win, "&Cancel",
                    19, 5, 10, 1,
                    IDCANCEL,
                    CWS_DEFCANCEL, CWS_NONE);
    ButtonSetClickedHook(ctrl, LoginDlgButtonHook, win);
    WindowCreate(ctrl);
}
```

Hingewiesen sei darauf, dass die Konstanten "IDC_EDUSERNAME" etc. innerhalb des Dialogmoduls z.B. per define angelegt sein müssen. Zudem ist erwähnenswert, dass der Dialog sich keine Zeiger auf die Kontrollelemente "merkt". Er kann später jederzeit über die ID des Kontrollelements auf dieses zugreifen:

```
ctrl = WindowGetCtrl(win, IDC_EDUSERNAME);
```

```
   if (ctrl)
   {
      EditGetText(ctrl, data->Username, 64);
   }
```

Da die Kontrollelemente Kindfenster des Dialogs sind, müssen sie nicht explizit nach dem Schließen des Dialogs zerstört werden. Dies geschieht automatisch dann, wenn das Dialogfenster selbst entfernt wird. Daher reduziert sich die Implementierung des DestroyHooks im vorliegenden Fall auf das Freigeben der Instanzdaten des Dialogs:

```
   static void
   LoginDlgDestroyHook(void* w)
   {
      free(((CUIWINDOW*) w)->InstData);
   }
```

5.5. Dialoge schließen

Ein Dialog wird über die Funktion "WindowClose" geschlossen, wobei an an diese Funktion der Result-Code übergeben wird, der später der Rückgabewert der Funktion "WindowModal" sein wird. Üblicherweise wird der Aufruf von "WindowClose" aus einem Callback-Hook eines Button- Kontrollelements heraus aufgerufen:

```
static void
LoginDlgButtonHook(void* w, void* c)
{
   CUIWINDOW* win = (CUIWINDOW*) w;
   CUIWINDOW* ctrl = (CUIWINDOW*) c;
   LOGINDLGDATA* data = (LOGINDLGDATA*) win->InstData;

   if (ctrl->Id == IDOK)
   {
      /* update data structures */
      WindowClose(win, IDOK);
   }
   else
   {
      WindowClose(win, IDCANCEL);
   }
}
```

Zu erwähnen ist dabei, dass "WindowClose" den Aufruf der Hook-Funktion "CanClose" bewirkt. Nur wenn das Dialogfenster und alle seine Kindfenster (falls sie diesen Hook überhaupt implementieren) einen Rückgabewert von "TRUE' zurückgeben, wird das Dialogfenster auch tatsächlich geschlossen.

Bevor ein Dialog mit "IDOK" (oder einem anderen Wert der einen Erfolg signalisiert) beendet wird, müssen die Kontrollelemente ausgelesen und in die Instanzdaten des Fensters

übertragen werden. Dabei erfolgt auch eine logische Prüfung der Daten in den Eingabefeldern. Notfalls kann man den Aufruf von "WindowClose" vermeiden, falls die Anwendereingaben unzufriedenstellend sind. Der Ort an dem eine solche Bearbeitung stattfinden sollte ist oben mit "update data structures" gekennzeichnet.

5.6. Vordefinierte Dialoge der libCUI

Damit häufig verwendete Dialoge, wie z.B. der Dialog zum Öffnen und Speichern von Dateien, nicht immer wieder neu implementiert werden müssen, wird eine Liste von Standarddialogen in der libCUI hinterlegt. Bislang ist die Auswahl noch recht bescheiden, wird aber in naher Zukunft wachsen.

Die bereits implementierten Dialoge werden hier kurz vorgestellt:

5.6.1. MessageBox

Die MessageBox wird verwendet um dem Anwender kurze Mitteilungen oder Fehlermeldungen anzuzeigen oder eine Auskunft von ihm zu verlangen, die sich mit "Yes", "No" oder mit "Retry", "Cancel" beantworten lässt.

Entgegen der üblichen Praxis wird die MessageBox nicht separat angelegt, erzeugt und anschließend modal ausgeführt. Vielmehr reicht ein einziger Funktionsaufruf aus, um den Dialog anzuzeigen und den Ergebniswert zurückzuliefern:

```
int MessageBox(CUIWINDOW* parent,
               const char* text,
               const char* title,
               int flags);
```

Als Vaterfenster kann ein beliebieges Fenster aus dem aktuellen Kontext angegeben werden, der Parameter "text" erhält die Textmeldung, im Parameter "title" wird der Fenstertitle übergeben und der Parameter "flags" steuert das Erscheinungsbild der MessageBox. Dabei können die folgenden Konstanten verwendet und mit dem Oder- Operator verknüpft werden:

Flag	Bedeutung
MB_NORMAL	MessageBox mit grauem Hintergrund (Standard)
MB_INFO	MessageBox mit einem cyanfarbigen Hintergrund (Info-Modus)
MB_ERROR	MessageBox mit einem roten Hintergrund (Fehler-Modus)
MB_OK	Nur eine Schaltfläche mit ”Ok’ Beschriftung (Standard). Der Rückgabewert des Dialogs ist immer IDOK.
MB_OKCANCEL	Zwei Schaltflächen beschriftet mit ”Ok” und ”Cancel”. Der Rückgabewert des Dialogs ist entweder IDOK oder IDCANCEL.
MB_YESNO	Zwei Schaltflächen beschriftet mit ”Yes” und ”No”. Der Rückgabewert des Dialogs ist entwerder IDYES oder IDNO.
MB_YESNOCANCEL	Drei Schalflächen beschriftet mit ”Yes”, ”No” und ”Cancel”. Der Rückgabewert ist entwerder IDYES, IDNO oder IDCANCEL.
MB_RETRYCANCEL	Zwei Schaltflächen beschriftet mit ”Retry” und ”Cancel”. Der Rückgabewert ist entwerder IDRETRY oder IDCANCEL.
MB_DEFBUTTON1	noch nicht implementiert.
MB_DEFBUTTON2	noch nicht implementiert.

Beispiele für Message Boxen:

```
MessageBox(win, "File saved successfully!", "Message", MB_OK);

MessageBox(win, "File not found!", "Error", MB_ERROR);

MessageBox(win, "I\nam\na\nmultiline\nMessage!",
           "Message", MB_OK);

if (MessageBox(win, "Really exit?",
               "Question", MB_YESNO) == IDYES)
{
   WindowClose(win, IDOK);
}
```

6. Hilfsbibliothek libCUI-util

Die libCUI-util ist eine Sammlung von Hilfsroutinen, die direkt oder indirekt für die Programmierung von CUI-Programmen nützlich sein können. Zudem finden sich hier eisfair-spezifische Funktionen zum Umgang mit Konfigurationsdateien und anderen administrativen Aufgaben. Keine der beiden Bibliotheken ist von der anderen abhängig, was bedeutet, dass Programme die Hilfsbibliothek verwenden können, ohne die libCUI zu verwenden zu müssen. Dies ist umgekehrt ebenso.

6.1. Verwenden der libCUI-util

Wie auch die libCUI, so kann auch die libCUI-util dynamisch und statisch gelinkt werden.

Dynamisch:

```
gcc -Wall -Wstrict-prototypes -o test test.c -lcui-util
```

Statisch:

```
gcc -o test test.c /usr/lib/libcui-util.a
```

6.2. XML-Parser

Der hier enthaltene Implementierung ist ein einfacher Parser, der XML-formatierte Dateien in eine Baumstruktur einliest. Der eingelesene Baum, der aus Knoten, Objekten und Attributen besteht, kann anschließend im Speicher durchsucht und ausgewertet werden. Zudem besteht die Möglichkeit, den eingelesenen Baum zu modifizieren und in modifizierter Form wieder in eine Datei zu schreiben.

In den Datei "cui-util.h" sind die folgenden Funktionsprototypen für den XML-Parser definiert:

```
XMLFILE* XmlCreate  (const char* filename);

void XmlDelete      (XMLFILE* xml);

void XmlSetErrorHook(XMLFILE* xml,
                     ErrorCallback errout,
                     void* instance);

void XmlAddSingleTag(XMLFILE* xml,
```

```
                      const char* name);

int XmlReadFile     (XMLFILE* xml);

int XmlWriteFile    (XMLFILE* xml);

void XmlPreserveNewline(XMLFILE* xml, int state);

XMLOBJECT* XmlGetObjectTree(XMLFILE* xml);
```

Mit der Funktion ”XmlCreate” wird eine Instanz der Stuktur ”XMLFILE” angelegt und dieser der Dateiname ”filename” zugewiesen. Nun kann mit ”XmlReadFile” die Datei eingelesen und mit ”XmlWriteFile” wieder zurückgeschrieben werden. Die Rückgabewerte sind endweder ”TRUE” oder ”FALSE” je nachdem, ob die Funktion erfolgreich abgeschlossen werden konnte oder nicht. Freigegeben wird die Datenstruktur ”XMLFILE” sowie der gesamte Objektbaum mit Hilfe der Funktion ”XmlDelete”.

Die Funktion ”XmlPreserveNewline” sorgt beim Einlesen der Datei dafür, dass Zeilenumbrüche erhalten bleiben, sich die Formatierung des Textes in der ASCII-Datei auch in den Daten des Objektbaums wiederfindet. Standardmäßig werden Zeilenumbrüche ignoriert.

Läuft der Parser beim Lesen der Datei auf einen Fehler, dann wird dieser über eine Callback-Funktion ausgegeben, die dem Parser zuvor über die Funktion ”XmlSetErrorHook” mitgeteilt werden muss. Der Paramter ”instance” kann dabei optional eine Fensterinstanz erhalten und ist speziell für das Zusammenspiel mit der libCUI gedacht. Die Callback-Funktion muss von Typ ”ErrorCallback” sein, die den folgenden Prototypen vorgibt:

```
typedef void (*ErrorCallback)(void* instance,
                              const char* errmsg,
                              const char* filename,
                              int line,
                              int is_warning);
```

Ein Code-Schnipsel, das eine XML-Datei einliest, kann nun wie folgt aussehen:

```
XMLFILE* xmldata = XmlCreate("myfile.xml");
XMLOBJECT* rootobj;
if (xmldata)
{
   XmlSetErrorHook(xmldata, MyErrorHook, win);
   if (XmlReadFile(xmldata))
   {
      rootobj = XmlGetObjectTree(xmldata);

      /* do something with the data */

   }
   XmlDelete(xmldata);
}
```

Die Implementierung der Callback-Funktion ist je nach Programm und Art der Fehlerbehandlung verschieden. Hier ein mögliches Beispiel:

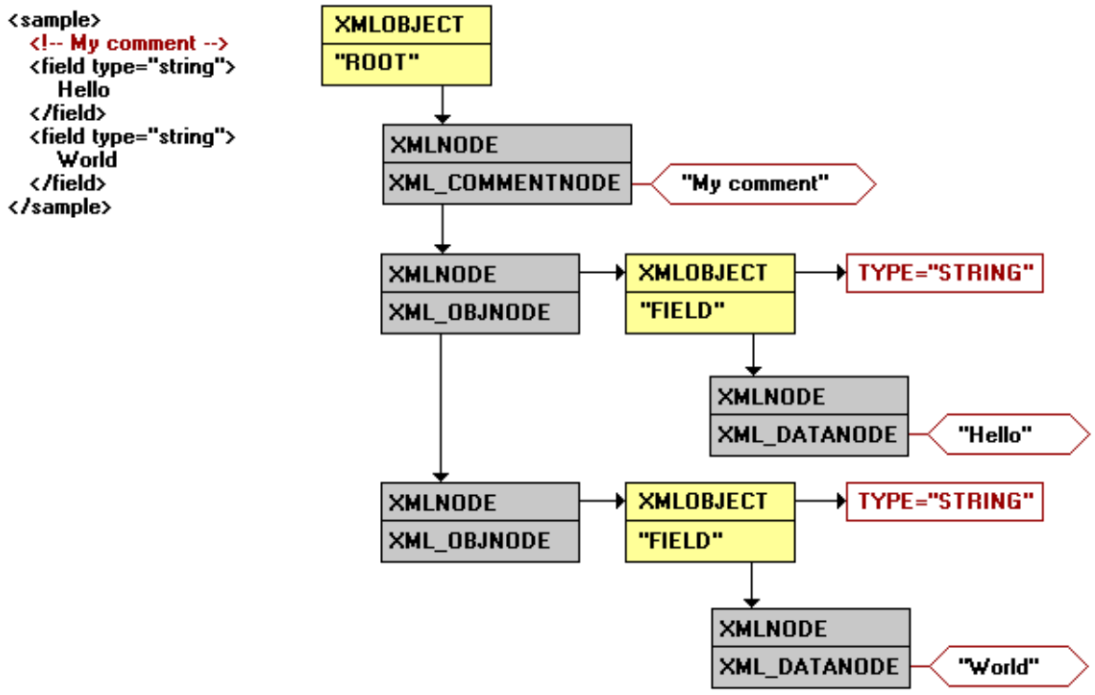
```
void
MyErrorHook(void* w, const char* errmsg, const char* filename,
             int linenr, int is_warning)
{
   CUIWINDOW* win = (CUIWINDOW*) w;
   if (win)
   {
      MYWINDATA* data = (MYWINDATA*) win->InstData;

      if ((data->NumErrors + data->NumWarnings) < 8)
      {
         char err[512];
         if (is_warning)
         {
            sprintf(err,"WARNING: (%i): %s",linenr,errmsg);
                        MywinAddMessage(win, err);
         }
         else
         {
            sprintf(err,"ERROR: (%i): %s",linenr,errmsg);
                        MywinAddMessage(win, err);
         }
      }
      else if ((data->NumErrors + data->NumWarnings) == 8)
      {
         MywinAddMessage(win, "... more errors");
      }

      if (is_warning)
      {
         data->NumWarnings++;
      }
      else
      {
         data->NumErrors++;
      }
   }
}
```

über "MywinAddMessage" wird die aktuelle Fehlermeldung zu einer Liste hinzugefügt (begrenzt auf max. acht Fehler). Nach Abschluss der Leseoperation kann später die Liste der Fehlermeldungen in einem Mitteilungsfenster (z.B. MessageBox) ausgegeben werden.

Die Abbildung zeigt, wie sich eine einfache XML-Datei im Objektbaum darstellt. Dieser besteht aus Objekten, Knoten, Attributen und Daten. Ausgehend vom stets vorhandenen Wur-



zelobjekt verläuft eine Liste von Knoten. Knoten können drei Typen besitzen:

XML_COMMENTNODE
Der Knoten steht für einen Kommentar. Das Datenfeld "Dataßeigt auf auf einen Kommentartext.

XML_OBJNODE
Der Knoten zeigt auf ein Objekt. Objekte besitzen selbst keine Daten, verfügen aber ihrerseits wieder über eine Liste von Knoten.

XML_DATANODE
Der Knoten steht für einen Block von Daten. Das Datenfeld "Dataßeigt auf auf eine Zeichenkette mit den enthaltenen Daten.

Die Elemente des Objektbaums bestehen aus den folgenden Datentypen:

```
typedef struct
{
   char* Name;        /* name of XML attribute */
   char* Value;       /* name of attribute value */
   void* Next;        /* next XML attribute */
} XMLATTRIBUTE;

typedef struct
{
   int   Type;        /* Type of node: data or child object */
```

```
   char* Data;       /* Data (if node is a data node) */
   int   DataLen;    /* Size of current data buffer */
   void* Object;     /* Pointer to child object (if object node) */

   void* Next;       /* Next node or NULL */
} XMLNODE;

typedef struct
{
   char* Name;              /* Name of XML object */

   XMLNODE*      FirstNode; /* The first node containing
                               data or a child object */
   XMLNODE*      LastNode;  /* The last node containing
                               data or a child object */
   XMLATTRIBUTE* FirstAttr; /* Pointer to first object attribute */
   XMLATTRIBUTE* LastAttr;  /* Pointer to last object attribute */
} XMLOBJECT;
```

Der Objektbaum kann nach dem Einlesen beliebig modifiziert werden. Die API Funktionen (deren Parameter weitestgehend selbsterklärend sein sollten) sind hier aufgelistet:

```
XMLOBJECT* XmlCreateObject    (XMLFILE* xml, XMLOBJECT* parent);
void       XmlDeleteObject    (XMLFILE* xml, XMLOBJECT* ob);
void       XmlSetObjectName   (XMLOBJECT* ob, const char* name);
void       XmlSetObjectData   (XMLOBJECT* ob, const char* data);
void       XmlAddObjectData   (XMLOBJECT* ob, const char* data);
void       XmlSetObjectComment(XMLOBJECT* ob, const char* data);
void       XmlAddObjectComment(XMLOBJECT* ob, const char* data);

XMLATTRIBUTE* XmlCreateAttribute  (XMLOBJECT* ob,
                                   const char* name);
void          XmlSetAttributeValue(XMLATTRIBUTE* attr,
                                   const char* name);
XMLATTRIBUTE* XmlGetAttribute     (XMLOBJECT* ob,
                                   const char* name);
```

6.3. Konfigurations-Parser

Der Konfigurationsparser ist ein Parser, der einfach aufgebaute Konfigurationsdateien der folgenden Form einliest:

```
FOO_NAME='Name'
FOO_PATH='/var/lib/foo'
```

```
FOO_ENUM_N='2'
FOO_ENUM_1_VALUE='1234'
FOO_ENUM_2_VALUE='0'
```

Die unter eifair gebräuchliche Form der Konfigurationsdateien sieht Wertepaare vor, bei denen der Wert in einfachen oder doppelten Anführungsstrichen steht. Es können auch ein- oder mehrdimensionale Arrays aufgebaut werden, indem Optionen einem Aufzählungsknoten (im Beispiel FOO_ENUM_N) zugeordnet werden.

Hinweis

Dieser Parser ist dazu gedacht, Konfigurationsdateien zu lesen, wie sie z.B. von CUI-Programmen unter eisfair verwendet werden (z.B. /etc/cui.conf). Der Nutzen des Parsers zum Einlesen von Paketkonfigurationen (z.B. /etc/config.d/foo) ist jedoch recht eingeschränkt, da Infrastrukturdateien (z.B. /etc/check.d/foo) nicht berücksichtigt werden. Dem Programm muss die Struktur der Konfigurationsdatei bekannt sein.

Der Parser bringt die folgende API zum Lesen der Dateien mit:

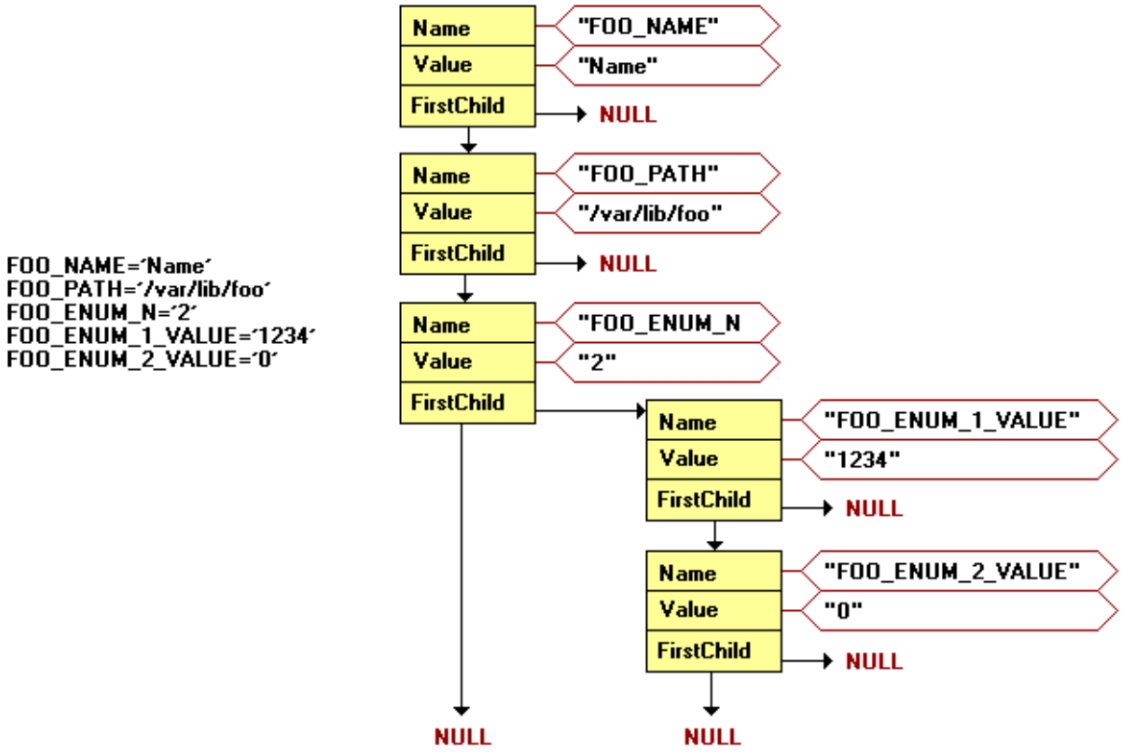
```
CONFIG*  ConfigOpen     (ErrorCallback errout,
                         void* instance);
void     ConfigAddNode  (CONFIG* cfg,
                         const char* n_node,
                         const char* mask);
void     ConfigReadFile (CONFIG* cfg,
                         const char* filename);
void     ConfigClose    (CONFIG* clg);
```

Der Aufruf von ConfigOpen bereitet das Lesen der Datei vor, indem eine "CONFIG" Struktur allokiert wird und dieser eine Callback- Funktion für Fehlerausgaben zugewiesen wird. Das Prinzip und der Funktionsprototyp der Callback- Funktion sind mit denen des XML-Parsers (siehe Abschnitt "XML-Parser") identisch.

Mit der Funktion "ConfigAddNode" wird dem Parser mitgeteilt, welche Optionen Array-Elemente sind und welchem Aufzählungsknoten sie zugeordnet werden. Der Parser lernt auf diese Weise die Struktur der Datei. Der Parameter "n_node" gibt dabei den Namen des Aufzählungsknotens an, der Parameter "mask" enthält eine Maske zur Erkennung des Optionsnamens.

Mit "ConfigReadFile" wird die Datei eingelesen. Es ist dabei zu beachten, dass die Funktion selbst keine Auskunft über Erfolg oder Misserfolg dibt. Fehler werden statt dessen über die Callback- Funktion ausgegeben und können hier ggf. gezählt werden.

Die Funktion "ConfigClose" gibt die "CONFIG" Struktur und alle damit verbundenen Daten wieder frei.



Nach dem Einlesen steht die Konfigurationsdatei als Wertebaum im Speicher zur Verfügung. Mit den folgenden API-Funktionen kann sie nun ausgewertet werden:

```
CONFENTRY*  ConfigGetEntry (CONFIG* cfg,
                            CONFENTRY* parent,
                            const char* name,
                            int * index);
const char* ConfigGetString(CONFIG* cfg,
                            CONFENTRY* parent,
                            const char* name,
                            OPT_TYPE type,
                            const char* defval,
                            int * index);
int         ConfigGetBool  (CONFIG* cfg,
                            CONFENTRY* parent,
                            const char* name,
                            OPT_TYPE type,
                            const char* defval,
                            int * index);
int         ConfigGetNum   (CONFIG* cfg,
                            CONFENTRY* parent,
                            const char* name,
                            OPT_TYPE type,
```

```
                                const char* defval,
                                int * index);
```

Die Funktionen dienen dem Auslesen unterschiedlicher Typen aus der Datei. Die Parameter haben dabei immmer die gleiche Bedeutung:

"cfg": CONFIG-Struktur der zuvor eingelesenen Datei.

"parent": Aufzählungsknoten oder NULL.

"name": Name der Option.

"type": Handelt es sich um einen optionalen Wert?

"defval": Standardwert, falls die Option nicht gefunden wird.

"index": Array mit Indexwerten im Falle von Arrays. Die Anzahl der Indexwerte im Array muss dabei der Dimension des Arrayelements "Option" entsprechen.

Bis auf "ConfigGetEntry" geben die Funktionen immer einen Wert zurück. Wird die Option nicht gefunden, dann wird der Standardwert zurückgegeben. Allerdings kann mit dem Parameter "type" (type = REQUIRED) vorgegeben werden, dass die Option existieren muss. In diesem Fall wird eine Meldung über die Callback- Funktion ausgegeben, falls der Wert nicht gefunden wurde.

Zum Lesen der oben aufgeführten Datei kann nun der folgende Code verwendet werden:

```
MYDATA* data = (MYDATA*) win->InstData;
CONFIG* cfg = ConfigOpen(ErrOut, win);
if (cfg)
{
   CONFENTRY* p;
   int i, index;

   /* reset error counter */
   data->NumErrors = 0;

   /* define file structure and read file */
   ConfigAddNode(cfg, "FOO_ENUM_N", "FOO_ENUM_%_VALUE");
   ConfigReadFile(cfg, "/etc/myconf.conf");

   /* get array dimensions */
   index = ConfigGetNum(cfg, NULL, "FOO_ENUM_N",
                        REQUIRED, "0", NULL);

   /* read array */
   p = ConfigGetEntry(cfg, NULL, "FOO_ENUM_N", NULL);
   if (p)
   {
      for (i = 0; i < index; i++)
```

```
      {
         val = ConfigGetNum(cfg, p, "FOO_ENUM_%_VALUE", &i);

         /* do something with "val" */
      }
   }
   ConfigClose(cfg);

   if (data->NumErrors > 0)
   {
      /* display errors */
   }
}
```

6.4. Co-Prozesse ausführen

Es ist oftmals erforderlich, andere Programme im Hintergrund zu starten und deren Textausgaben oder Ergebniswerte anschließend weiterzuverarbeiten. Damit lassen sich Aufgaben an andere Programme delegieren, Schnittstellen zu Scripten etc. schaffen und vorhandene Funktionalität nutzen.

Die libCUI-util stellt eine API-Funktion zur Verfügung, mit der Co-Prozesse ausgeführt werden können:

```
int  RunCoProcess(const char* filename,
                  char* const parameters[],
                  TextCall callback,
                  int* exitcode);
```

Der Parameter ”filename” gibt den Namen und den Pfad des externen Programms an, das Array ”parameters” enthält die Programmargumente, ”callback” ist eine Callback-Funktion für die Textausgaben des Programms und ”exitcode” ist die Variable die den Rückgabewert des aufgerufenen Programms zurückgibt. Der Rückgabewert der Funktion ”RunCoProcess” ist ”TRUE” wenn der Aufruf erfolgreich war und ”FALSE” falls nicht.

Bei ”parameters” ist zu berücksichtigen, dass das erste Element (parameters[0]) den Namen des Programms erhalten muss. Das letzte Element enthält einen Nullzeiger und signalisiert damit das Ende der Liste.

Die Funktion vom Typ ”TextCall” muss dem folgenden Prototypen entsprechen:

```
typedef void (*TextCall) (const char* buffer, int source);
```

Sie erhält die Textausgaben des Programms zeilenweise (also immer dann, wenn ein Zeilenumbruch vom Programm zurückgegeben wird. Der Parameter "source" kann die Werte "PIPE_STDOUT" und "PIPE_STDERR" annehmen und gibt damit an, über welchen Augabekanal der Text ausgegeben wurde.

Die Funktion "RunCoProcess" hält das Programm im Vordergrund an, bis der CoProcess beendet wurde.

Ein Beispiel:

```
void RunCallback(const char* buffer, int source)
{
   if (source == PIPE_STDOUT)
   {
      printf("INFO:  %s\n", buffer);
   }
   else
   {
      printf("ERROR: %s\n", buffer);
   }
}

int RunScript(const char* scriptfile)
{
   int result, status;
   const char* p[3];

   /* initialize for interpreted scripts */
   p[0] = "/bin/bash";
   p[1] = scriptfile;
   p[2] = 0;

   /* execute */
   result = RunCoProcess("/bin/bash", (char**) p,
                         RunCallback, &status);

   /* evaluate result */
   return (result == TRUE) && (status == 0);
}
```

ToDo: Hier sollte noch ein Instanzzeiger enthalten sein, damit dieser an die Callback-Routine übergeben werden kann.

Ein Hinweis sein hier auch nochmal auf das Terminal Kontrollelement der libCUI gegeben. Dieses erlaubt die Ausführung von Programmen in einem Terminal-Fenster, während der Rest der Anwendung normal weiterläuft.

7. Scripting

Damit auch Shell-Skripte mit einer interaktiven CUI Benutzerschnittstelle versehen werden können, wurde die libCUI um die Funktion einer Shell-API erweitert. Dabei wurde bewusst auf die Einbindung einer Skript-Sprache wie Perl oder Python verzichtet, da damit die Integration in ein Basissystem, wie das von eisfair, problematisch wäre. Statt dessen stehen die Funktionen direkt für die Bash zur Verfügung, die als Hintergrundprozess von einem Frontend-Programm ausgeführt wird.

Die Programmierung entspricht in weiten Teilen der CUI-Programmierung in der Sprache C. Darum ist es in jedem Fall ratsam, das Kapitel "Elementare Fenstertechnik" gelesen zu haben. Zudem wird hier lediglich eine prinzipielle Beschreibung der Einbindung von Shell-Skripten geliefert. Eine Refenrez aller API-Funktionen befindet sich im Anhang A.

7.1. Funktionsprinzip

Damit ein Shell-Skript eine CUI-Oberfläche benutzen kann, muss es als Hintergrundprozess von einem Frontend-Programm ausgeführt werden. Das Frontend für allgemeine Skripte heißt "shellrun.cui", aber auch der Konfigurationseditor, "edit-conf.cui", kann Skripte als Co-Prozess ausführen.

Im Falle von "shellrun.cui" wird das jeweilige Skript dem Frontend-Programm als Argument übergeben. Man kann damit z.B. Curses-Menüs zur Auswahl von Funktionen o.ä. realisieren, wie sie häufig Paketentwicklung vorkommen. Man kann aber auch Oberflächen schreiben, mit denen der Anwender seine Daten pflegen oder Dateien auswählen kann. Generell ist hier nahezu alles denkbar, was mit einem Shell-Skript machbar ist.

Auch der Konfigurationseditor (allgemein als ECE bezeichnet) verfügt über die Möglichkeit Shell-Skripte auszuführen. Dies dient dazu, die Eingabezeile im ECE gegen einen Dialog zu ersetzen, der z.B. eine Auswahlliste für Werte anbietet oder die Benutzereingabe auf andere Weise erleichtert. Welches Skript aufgerufen werden soll ermittelt das Programm dabei aus der Paketkonfiguration.

Das Frontend-Programm erzeugt nach dem Programmstart zwei Named-Pipes für die Client-Server Kommunikation ($HOME/.cui/<proc-id>rp und $HOME/.cui/<proc-id>wp) und erwartet von dem Shell-Skript, dass dieses die Pipes öffnet und eine Verbindung initiiert. Dazu gibt es bereits vorbereitete Funktionen die in das Shell-Skript auf einfache Weise eingebunden werden können.

Ist die Verbindung einmal hergestellt, dann fungiert das Frontend als eine Art Window-Manager (Server), der vom Shell-Skript (Client) API-Befehle z.B. zur Konstruktion von Fen-

stern entgegen nimmt. Letzteres wiederum wird vom Frontend mit Informationen über Benutzereingaben in Form von Call-Back Routinen benachrichtigt.

Der Programmablauf in der Shell erfolgt nicht mehr, wie gewohnt, in linearer Form, sondern ist Ereignisorientiert, wie man das von der Programmierung graphischer Oberflächen her kennt. Diese Vorgehensweise ist für einen routinierten Skript-Programmierer anfänglich sicher etwas ungewohnt.

Der grundlegende Aufbau eines CUI-fähigen Shell-Skripts sieht wie folgt aus:

```
#!/bin/sh

. /var/install/include/cuilib
[ evtl. weitere includes ]

[ Skript-Funktionen ]

cui_init
cui_run

exit 0
```

Die vollständige Funktionalität des Skriptes steckt in Funktionen, die vom Frontend aufgerufen werden. Der Programmfluss läuft linear bis zu der Funktion "cui_run" durch und bleibt dort ein einer Schleife hängen, die ständig die Eingabe-Pipe überwacht. Trifft dort ein Funktionsaufruf vom Frontend ein (Ereignis), dann wird eine Funktion unter dem entsprechenden Namen aufgerufen. Diese sollte natürlich unbedingt im Skript existieren! Da bis auf die Einstiegsfunktioin alle Ereignisroutinen vom Shell-Programmierer explizit beim Frontend angemeldet werden, ist das aber kein Problem. Wichtig ist jedoch, dass die Funktionen oberhalb von "cui_run" und nach dem Source-Include der cuilib implementiert sind.

Welche Funktion vom Frontend zuerst aufgerufen wird (Einstiegsfunktion), ist vom jeweiligen Programm abhängig. Bei shellrun.cui heißt sie "init" beim ECE werden nacheinander die Funktionen "setdata", "exec_dialog" und "getdata" aufgerufen. In jedem Fall beendet sich die Warteschleife in "cui_run", wenn die Funktion "exit" aufgerufen wird (die nicht implementiert sein muss / darf).

Eine Funktion, die vom Frontend aufgerufen wurde, muss unbedingt mit "cui_return" beendet werden. Anderenfalls wartet das Frontend vergeblich auf eine Rückmeldung, was ein beliebter Fehler ist, nach dem man beliebig lange suchen kann. Die übergabeparameter vom Frontend an das Skript sind in den Variablen $p2..$pn gespeichert (das ist für die jeweilige Funktion fest definiert). Der Rückgabewert wird an "cui_return" übergeben.

Innerhalb der Funktionen, die vom Frontend aus aufgerufen werden, kann das Skript wiederum API-Funktionen des Frontends nutzen. Diese sind in der Referenz im Anhang aufgeführt und dienen zum Anlegen und Steuern von Kontrollelementen und vielem mehr.

Der Rückgabewert der API-Script-Funktion ($?) zeigt an, ob der Aufruf erfolgreich war oder ob während der Verarbeitung etwas schiefgegangen ist. Bei den meisten Funktionen kann man ihn ignorieren. Die Rückgabewerte des Frontends wiederum sind wiederum in den Variablen $p2..$pn zu finden und für die jeweilige API-Funktion genau definiert.

Beispiel:

```
local ctrl="0"

cui_window_getctrl "$win" "$IDC_COLORS"
if [ "$p2" != "0" ]
then
    ctrl="$p2"     # gefunden!
fi
```

sucht das Kindfenster von $win mit der ID $IDC_COLORS. Der Rückgabewert in $p2 ist das Window-Handle des Kindfensters oder 0, falls dieses nicht gefunden wurde.

Wichtig ist, dass man in Funktionen, wann immer es möglich ist, mit lokalen Variablen arbeitet. Dafür kennt die bash das Schlüsselwort "local". Anderenfalls riskiert man, dass die gleichlautende Variable in einer anderen Funktion unbemerkt überschrieben wird.

7.2. Ablaufverfolgung

Die Ausgabe von Text über die Standardausgabe ist unter dem Curses-Frontend (z.B. mit echo) nicht mehr möglich. Für Debug Zwecke gibt es jedoch den Kommandozeilenschalter −−debug über den sowohl die Frontend-Backend Kommunikation, als auch die Standardausgabe in die Datei /tmp/outcui.log ausgegeben wird. Mit

less +F /tmp/outcui.log

erhält man eine Art Trace-Output, über den der Programmablauf in Echtzeit verfolgt werden kann.

Der Trace ist vor allem immer dann interessant, wenn man verfolgen möchte, welche API-Funktionen und Callbacks in welcher Reihenfolge aufgerufen werden, da auf diese Weise sichtbar wird wo sich ggf. die Kommunikation aufgehängt oder das Backend-Skript beendet hat.

Ein Ausschnitt aus einem typischen Trace sieht beispielsweise wie folgt aus:

```
-> H    init
<- C    1               0       0       0       0       45058
-> R    0       134624472
<- C    32      134624472       DESKTOP
-> R    0
<- C    40      134624472       Eisfair User Manager
-> R    0
<- C    44      134624472       Commands: F10=Exit
-> R    0
<- C    45      134624472       V1.0.0
-> R    0
...
<- H    1
```

In der ersten Spalte steht in welche Richtung die Daten geflossen sind und um welche Art von Aufruf es sich handelt. Dies entspricht der folgenden Tabelle:

Aufruf	Bedeutung
-> H	Der Server Ruft eine Hook-Funktion auf. Der Name der Funktion steht im ersten Parameter.
<- H	Der Client beendet eine Hook-Funktion mit "cui_return". Der Rückgabewert steht im ersten Parameter.
<- C	Der Client ruft eine API-Funktion auf. Die Nummer der Funktion steht im ersten Parameter.
-> R	Das Frontend beendet eine API-Funktion. Der erste Parameter ist bei Erfolg immer 0, dann folgen die Rückgabewerte.

API-Aufrufe werden durch ihre Nummer repräsentiert. Die entsprechenden Nummern können in der Datei "/var/install/include/cuilib" nachgelesen werden. Für das Beispiel oben finden sich z.B. die folgenden Definitionen in der Datei:

API_WINDOWNEW = 1
API_WINDOWCOLSCHEME = 32
API_WINDOWSETTEXT = 40
API_WINDOWSETLSTATUSTEXT = 44
API_WINDOWSETRSTATUSTEXT = 45

7.3. Scriptfähige Anwendungen und deren API

7.3.1. shellrun.cui

Mit "shellrun.cui" lassen sich eigenständige interaktive CUI Skript-Programme realisieren. Die Einstiegsfunktion die "shellrun.cui" im Skript-Programm zwingend voraussetzt heißt "init()". Alles weitere hängt davon ab, wie die Anwendung in der Initialisierungsfunktion eingerichtet wurde.

Das wohl einfachste Programm für "shellrun.cui" sieht folgendermaßen aus:

```
#!/bin/sh

. /var/install/include/cuilib

#-----------------------------------------------------------------
# init routine (entry point of all shellrun.cui based programs)
#    $p2 --> desktop window handle
#-----------------------------------------------------------------

init()
{
```

```
    cui_message "$p2" "Hallo Welt" "Hallo" "$MB_OK"
    cui_return 0
}

#-----------------------------------------------------------------
# main routines (always at the bottom of the file)
#-----------------------------------------------------------------

cui_init
cui_run

exit 0
```

Im vorliegenden Fall wird in der init-Routine lediglich ein Mitteilungsfenster angezeigt. Anschließend wird die Routine beendet, ohne dass weitere Fenster erzeugt werden. Gibt es nach Beenden von ”init()” kein aktives Fenster mehr, dann beendet sich ”shellrun.cui” automatisch - wurden dagegen Fenster angelegt, dann läuft das Programm weiter, bis die Funktion ”cui_window_quit” explizit aufgerufen wird.

Der Aufruf von ”cui_message” öffnet ein modales Mitteilungsfenster und zeigt darin den angegebenen Text an. Der Programmfluss bleibt bei ”cui_message” stehen, bis der Anwender das Fenster mit OK schließt. Die Definition der API-Funktion ”cui_message” kann der Referenz entnommen werden.

Wurde das Programm unter dem Namen ”sayhallo.sh” abgespeichert, dann gibt es zwei Möglichkeiten es aufzurufen - direkt und indirekt:

```
eis # /var/install/bin/shellrun.cui ./sayhallo.sh
```

oder

```
eis #./sayhallo.sh
```

im zweiten Fall sorgt das Skript selbst für den Aufruf von ”shellrun.cui”.

Ein für ”shellrun.cui” erstelltes Skript-Programm kann selbstverständlich um einiges komplexer sein als das hier gezeigte Beispiel. Für eine weiterführende Beschreibung wird an anderer Stelle ein Tutorial entstehen.

7.3.2. edit-conf.cui

Der Konfigurationseditor ”edit-conf.cui” zeigt im Normalfall einen Dialog mit einer einfachen Eingabezeile um einen Wert in der Konfiguration zu bearbeiten. Dieses Verhalten läßt sich mit einem Shell-Skript ändern, das im Verzeichnis ”/var/install/dialog.d” abgelegt wird. Man kann auf diese Weise Dialoge schreiben, die beispielsweise Auswahllisten anzeigen oder komplexe Werte aus mehreren Eingabeelementen zusammensetzen.

Soll eine Konfigurationsvariable bearbeitet werden, dann sucht der Editor im Verzeichnis ”/var/install/dialog.d”, ob dort eine Datei liegt, die sich einer Prüfregel der Variablen aus

der Datei "/etc/check.d/$package" zuordnen läßt. Wird eine solche Datei gefunden, dann wird sie als Shell-Skript ausgeführt.

Beispiel:

Eine Zeile in der Datei "/etc/check.d/foo"...

```
FOO_VALUE       -           -         FOO_EDIT
```

...veranlasst den Editor im Verzeichnis "/var/install/dialog.d" nach einer Datei mit dem Namen "FOO_EDIT.sh" zu suchen. Ist sie vorhanden, dann wird sie als Backend-Skript ausgeführt. Anderenfalls wird der Standarddialog verwendet.

Das Programm "edit-conf.cui" erwartet drei Einstiegsfunktionen im Skriptprogramm: "setdata()", "exec_dialog()" und "getdata()". Zunächst wird die Funktion "setdata()" aufgerufen, über die dem Skript der aktuelle Wert der zu ändernden Variablen mitgeteilt wird. Anschließend führt "edit-conf.cui" die Funktion "exec_dialog()" aus, die den Dialog zur Bearbeitung des Wertes anzeigt. Wird "exec_dialog()" mit dem Rückgabewert "$IDOK" beendet, dann wird der geänderte Wert über "getdata()" abgerufen, geprüft und der Variablen zugewiesen. Ein Rückgabewert von "$IDCANCEL" dagegen führt dazu, dass die Bearbeitung ohne Zuweisung abgebrochen wird.

Ein einfaches (und zugleich völlig sinnloses) Beispielprogramm sieht wie folgt aus:

```
#!/bin/sh

. /var/install/include/cuilib

value=""

#-----------------------------------------------------------------
# setdata routine (set value to be modified)
#    $p2 --> value
#-----------------------------------------------------------------

setdata()
{
    value="$p2"
    cui_return 1
}

#-----------------------------------------------------------------
# getdata routine (return modified value)
#-----------------------------------------------------------------

getdata()
{
    cui_return "$value"
}
```

```
#-----------------------------------------------------------------
# exec_dialog routine (show edit dialog)
#    $p2 --> parent window handle
#    $p3 --> name of config variable
#-----------------------------------------------------------------
exec_dialog()
{
    local dlg="$p2"

    cui_message "$dlg" "Wert wirklich "andern?" "Frage" "$MB_YESNO"
    if [ "$p2" == "$IDYES" ]
    then
        value="ge"andert"
        cui_return "$IDOK"
    else
        cui_return "$IDCANCEL"
    fi
}

#-----------------------------------------------------------------
# main routines (always at the bottom of the file)
#-----------------------------------------------------------------

cui_init
cui_run

exit 0
```

Die Programmierung von Dialogen für den ECE läßt sich vereinfachen, indem die Include-Datei "/var/install/include/ecelib" in das Skript eingebunden wird. Sie implementiert die Funktionen "getdata()" und "setdata()" und bietet ein paar Standarddialoge für Auswahllisten und ähnliche Standardaufgaben.

ece_select_list_dlg

Der in der ecelib definierte Dialog "ece_select_list_dlg" implementiert eine einfache Auswahlliste, die zentriert über dem Editorfenster angezeigt wird. Der Anwender kann dabei aus einer Reihe möglicher Werte wählen und den gewünschten Wert (z.B. mit Enter) bestätigen. Die Auswahl wird dann in die bearbeitete Konfigurationsvariable des ECE übertragen.

Eine mögliche Verwendung des Dialogs ist im folgenden Listing dargestellt:

```
#!/bin/sh

. /var/install/include/cuilib
```

```
. /var/install/include/ecelib

#----------------------------------------------------------------
# exec_dailog
# ece --> request to create and execute dialog
#         $p2 --> main window handle
#         $p3 --> name of config variable
#----------------------------------------------------------------
exec_dialog()
{
    local win="$p2"

    sellist="BLACK,RED,GREEN,BROWN,BLUE,MAGENTA,CYAN,LIGHTGRAY,"
             DARKGRAY,LIGHTRED,LIGHTGREEN,YELLOW,LIGHTBLUE,
             LIGHTMAGENTA,LIGHTCYAN,WHITE"

    ece_select_list_dlg "$win" "Colors" "$sellist"
}

#----------------------------------------------------------------
# main routine
#----------------------------------------------------------------

cui_init
cui_run

#----------------------------------------------------------------
# end
#----------------------------------------------------------------

exit 0
```

Bei diesem Beispiel wird folgende Auswahlliste ausgegeben:

```
+----------[ Colors ]-----------+
|                               |
| +-------------------------+^| |
| | BLACK                   | | |
| | RED                     | | |
| |<GREEN ================>| | |
| | BROWN                   | | |
| | BLUE                    | | |
| | MAGENTA                 | | |
| | CYAN                    | | |
| +-------------------------+v+ |
|     [< OK >] [ Cancel ]       |
|                               |
```

```
+------------------------------+
```

Zu beachten ist, dass die Funktion ”ece_select_list_dlg” die Behandlung des Rückgabewertes an den ECE übernimmt. Deshalb ist der Aufruf von ”cui_return” in der Funktion ”exec_dialog” nicht mehr nötig und auch nicht mehr erlaubt.

ece_comment_list_dlg

Der Dialog ”ece_comment_list_dlg” ist eine Erweiterung des ”ece_select_list_dlg” Dialogs. Hier ist es möglich neben den Auswahloptionen Kommentare anzugeben, die gemeinsam mit den Auswahlwerten in einer zweispaltigen Auswahlliste dargestellt werden. Auf diese Weise sieht der Anwender nicht nur die Optionen (die für sich gesehen u.U. vergleichsweise nichtssagend sind), sondern auch einen erklärenden Text, der die Auswahl erleichtert.

Ein Beispiel ist im folgenden Listing dargestellt:

```
#!/bin/sh

. /var/install/include/cuilib
. /var/install/include/ecelib

#-----------------------------------------------------------------
# exec_dailog
# ece --> request to create and execute dialog
#         $p2 --> main window handle
#         $p3 --> name of config variable
#-----------------------------------------------------------------
exec_dialog()
{
    local win="$p2"

    sellist="option1|Dies ist Option1,option2|Dies ist Option2,
             option3|Dies ist Option3,option4|Dies ist Option4"

    ece_comment_list_dlg "$win" "Optionen" "$sellist"
}

#-----------------------------------------------------------------
# main routine
#-----------------------------------------------------------------

cui_init
cui_run

#-----------------------------------------------------------------
# end
#-----------------------------------------------------------------
```

```
exit 0
```

Bei diesem Beispiel wird folgende Auswahlliste ausgegeben:

```
+-------------[ Optionen ]----------------+
|                                         |
| +-----------------------------------+^| |
| | option1    | Dies ist Option1     | | |
| | option2    | Dies ist Option2     | | |
| |<option3 ===| Dies ist Option3 ===>| | |
| | option4    | Dies ist Option4     | | |
| +-----------------------------------+v+ |
|           [< OK >] [ Cancel ]           |
|                                         |
+-----------------------------------------+
```

Wie schon beim vorhergehenden Beispiel darf die Funktion ”exec_dialog” auch hier nicht mit einem Aufruf von ”cui_return” abgeschlossen werden, da dies bereits im Dialog selbst erfolgt ist.

ece_select_cblist_dlg

Der in der ecelib definierte Dialog ”ece_select_cblist_dlg” implementiert eine Auswahlliste unter Nutzung von Check Boxen, die zentriert über dem Editorfenster angezeigt wird. Der Anwender kann dabei einer Reihe möglicher Werte unter Nutzung der Cursortasten ansteuern und über die Space Taste auswählen. Ausgewählte Werte werden mit [x] markiert. Nicht ausgewählte Werte haben die Markierung [ ]. Die gewünschten Werte können (z.B. mit Enter) bestätigen werden. Die Auswahl wird dann in die bearbeitete Konfigurationsvariable des ECE übertragen.

Beispiel für die Verwendung des Dialogs ece_select_cblist_dialog :

```
#!/bin/sh

. /var/install/include/cuilib
. /var/install/include/ecelib

#-----------------------------------------------------------------
# exec_dailog
# ece --> request to create and execute dialog
#         $p2 --> main window handle
#         $p3 --> name of config variable
#-----------------------------------------------------------------
exec_dialog()
{
  local win="$p2"
```

```
  sellist='apache2,ldapserver,mail,mini_httpd,partimg,pure-ftpd,ssmtp'

  ece_select_cblist_dlg "$win" \
      "Select one or more elements from a list" "$sellist"
}

#----------------------------------------------------------------
# main routine
#----------------------------------------------------------------

cui_init
cui_run

#----------------------------------------------------------------
# end
#----------------------------------------------------------------

exit 0
```

Bei diesem Beispiel wird folgende Auswahlliste ausgegeben:

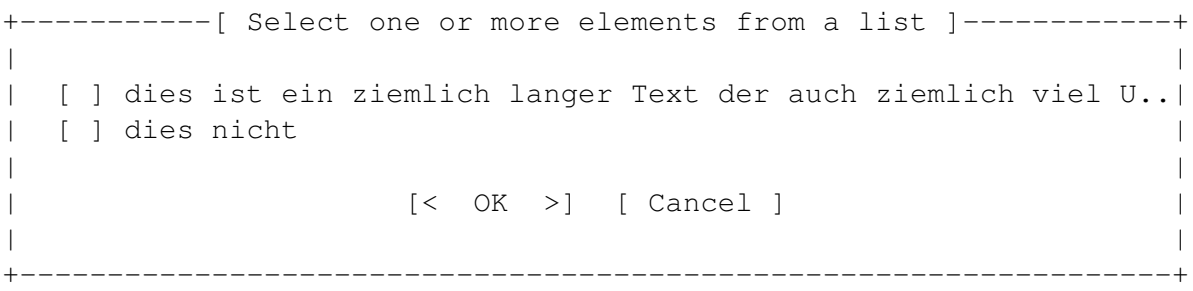
```
+-[ Select one or more elements from a list ]-+
|                                              |
|  [x] apache2                                 |
|  [ ] ldapserver                              |
|  [ ] mail                                    |
|  [ ] mini_httpd                              |
|  [ ] partimg                                 |
|  [x] pure-ftpd                               |
|  [ ] ssmtp                                   |
|                                              |
|           [<  OK  >]  [ Cancel ]             |
|                                              |
+----------------------------------------------+
```

Für maximal 15 Elemente ist Platz auf einem 80/25 Display, daher wird bei mehr als 15 Elemente eine Fehlermeldung erzeugt:

```
+----[ Checkbox selection ]----+
|                              |
|  Too many elements (16/15).  |
|                              |
|          [<  OK  >]          |
+------------------------------+
```

Es können selbstverständlich auch längere Elemente angezeigt und ausgewählt werden, allerdings ist die Ausgabe auf 58 Zeichen begrenzt. Die restlichen Zeichen werden durch ein .. angedeutet.

Beispiel:



ACHTUNG: Es wird nicht geprüft, ob ein Element mehrfach vorkommt.

Mittels der Variablen ECE_SELECT_CBLIST_VAL_SEPARATOR kann der Separator der Select-Liste eingestellt werden.

ECE_SELECT_CBLIST_VAL_SEPARATOR=':'
setzt z.B. den Separator auf den Doppelpunkt.

8. libCUI unter eisfair

Es gibt ein paar Gemeinsamkeiten und Richtlinien, die für CUI-Programme in der eisfair-Umgebung gelten. Diese werden hier angesprochen.

8.1. Farben und Konfiguration

Einstellungen, die die Darstellung und Funktion von libCUI- Programmen beeinflussen, werden der Bibliothek zur Laufzeit mitgeteilt. Dies sind z.B. die Einstellung für den Farbmodus sowie die Mauseingabe, die als Parameter an die Funktion "WindowStart" übergeben werden. Auch dazu gehören die Farbprofile, die über "WindowAddColScheme" gesetzt werden.

8.1.1. Aufbau der Konfigurationsdateien

CUI-Programme unter eisfair, die auf der libCUI- Bibliothek aufsetzen, verwenden als gemeinsame Konfiguration die Datei "/etc/cui.conf". Dort sind die Optionen hinterlegt, die allen Programmen gemeinsam sind. benötigt ein Programm spezifische Einstellungen, dann sollte es eine eigene Konfigurationsdatei anlegen, die den Namen "«progname».cui.conf" trägt. Beispiel: "/etc/myprog.cui.conf".

Die Datei "/etc/cui.conf" hat den folgenden Aufbau (gekürzt):

```
#-------------------------------------
# global options
#-------------------------------------

CUI_USE_COLORS='yes'
CUI_USE_MOUSE='no'

#-------------------------------------
# color scheme "WINDOW"
#-------------------------------------

CUI_WINDOW_WND_COLOR='BLUE'
CUI_WINDOW_WND_SEL_COLOR='LIGHTGRAY'
CUI_WINDOW_WND_TXT_COLOR='LIGHTGRAY'
CUI_WINDOW_SEL_TXT_COLOR='BLACK'
```

```
CUI_WINDOW_INACT_TXT_COLOR='DARKGRAY'
CUI_WINDOW_HILIGHT_COLOR='YELLOW'
CUI_WINDOW_TITLE_TXT_COLOR='BLACK'
CUI_WINDOW_TITLE_BKG_COLOR='LIGHTGRAY'
CUI_WINDOW_STATUS_TXT_COLOR='BLACK'
CUI_WINDOW_STATUS_BKG_COLOR='LIGHTGRAY'

#-------------------------------------
# color scheme "DESKTOP"
#-------------------------------------

...
```

Neben dem Farbschema "WINDOW" gibt es noch die Standardschemata "DESKTOP", "DIALOG", "MENU", "TERMINAL" und "HELP". Die Namen der Optionen enthalten dann anstelle von "WINDOW" den Namen des jeweiligen Schemas. Beispiel: "CUI_HELP_WND_COLOR" anstelle von "CUI_WINDOW_WND_COLOR".

8.1.2. Verarbeiten von Konfigurationsdateien

Die libCUI sieht aus Gründen der Verallgemeinerung keinen Code vor, der ein automatisches Einlesen der Konfiguration erlaubt. Vielmehr ist das jeweils vom Hauptprogramm der Anwendung zu leisten, bevor der Fenstermodus aktiviert wird.

Ein Beispiel, das durchaus als Template verwendet werden kann, sieht wie folgt aus:

Zunächst erstmal die Funktion "main", die das Einlesen der Konfiguration veranlasst, falls die Datei "etc/cui.conf" existiert:

```
int
main(int argc, char* argv[])
{
   CUIWINDOW* window;

   /* read config file */
   if (FileExists("/etc/cui.conf"))
   {
      ReadConfig("/etc/cui.conf");
   }

   ...

   /* start cui subsystem */
   WindowStart(UseColors, UseMouse);
   atexit(quit);
```

```
   ...

   return WindowRun();
}
```

Dann die Funktion ”ReadConfig”, die mit Hilfe des Config-Parsers der libCUI die Datei einliest:

```
static int
ReadConfig(const char* filename)
{
   CONFIG*     cfg;
   CUIWINCOLOR colrec;

   cfg = ConfigOpen(ErrorOut, NULL);

   NumErrors = 0;
   NumWarnings = 0;
   ConfigReadFile(cfg, filename);

   UseColors = ConfigGetBool(cfg,
                             NULL,
                             "CUI_USE_COLORS",
                             REQUIRED,
                             "yes",
                             NULL);
   UseMouse  = ConfigGetBool(cfg,
                             NULL,
                             "CUI_USE_MOUSE",
                             REQUIRED,
                             "no",
                             NULL);
   if (ReadColorRec(cfg, "WINDOW", &colrec))
   {
      WindowAddColScheme("WINDOW", &colrec);
   }
   if (ReadColorRec(cfg, "DESKTOP", &colrec))
   {
      WindowAddColScheme("DESKTOP", &colrec);
   }

   ...

   ConfigClose(cfg);

   if ((NumErrors != 0) || (NumWarnings != 0))
```

```
   {
      char tmp;

      printf("%i error(s), %i warning(s)\n",
             NumErrors, NumWarnings);
      printf("file: %s", "/etc/cui.conf\n");

      printf("\nPress ENTER to continue\n");
      scanf("%c", &tmp);
   }

   return (NumErrors == 0);
}
```

die Variablen ”NumErrors” und ”NumWarnings” sind global angelegte Ganzzahlenvariablen von Typ ”int”. Ebenso die Variablen ”UseColor” und ”UseMouse”. Das Einlesen von Farbprofilen wird an die Funktion ”ReadColorRec” delegiert, der der Name des Profils übergeben wird. Im Sinne der übersichtlichkeit wurde die Liste der Farbprofile auf zwei gekürzt.

Die Funktion ”ReadColorRec” hat den folgenden Aufbau:

```
static int
ReadColorRec(CONFIG* cfg, const char* name,
             CUIWINCOLOR* colrec)
{
   char option[128 + 1];

   sprintf(option, "CUI_%s_WND_COLOR", name);
   if (ConfigGetEntry(cfg, NULL, option, NULL))
   {
      colrec->WndColor = TranslateColor(cfg,
                                        option,
                                        "BLUE");

      sprintf(option, "CUI_%s_WND_SEL_COLOR", name);
      colrec->WndSelColor = TranslateColor(cfg,
                                           option,
                                           "LIGHTGRAY");

      sprintf(option, "CUI_%s_WND_TXT_COLOR", name);
      colrec->WndTxtColor = TranslateColor(cfg,
                                           option,
                                           "LIGHTGRAY");

      ...
```

```
      return TRUE;
   }
   return FALSE;
}
```

Wie zu sehen ist, wird hier der zur jeweiligen Farboption passende Optionsname gebildet und dann an die Funktion "TranslateColor" übergeben. Erneut wurde das Listing zur besseren übersicht gekürzt.

Zu guter Letzt nun noch das Listing der Funktion "TranslateColor":

```
static int
TranslateColor(CONFIG* cfg, const char* option,
               const char* defval)
{
   int col = BLACK;
   const char* color;

   color = ConfigGetString(cfg,
                           NULL,
                           option,
                           OPTIONAL,
                           defval,
                           NULL);

   if (strcasecmp(color,"BLACK") == 0)
      col = BLACK;
   else if (strcasecmp(color,"RED") == 0)
      col = RED;
   else if (strcasecmp(color,"GREEN") == 0)
      col = GREEN;
   else if (strcasecmp(color,"BROWN") == 0)
      col = BROWN;
   else if (strcasecmp(color,"BLUE") == 0)
      col = BLUE;
   else if (strcasecmp(color,"MAGENTA") == 0)
      col = MAGENTA;
   else if (strcasecmp(color,"CYAN") == 0)
      col = CYAN;
   else if (strcasecmp(color,"LIGHTGRAY") == 0)
      col = LIGHTGRAY;
   else if (strcasecmp(color,"DARKGRAY") == 0)
      col = DARKGRAY;
   else if (strcasecmp(color,"LIGHTRED") == 0)
      col = LIGHTRED;
   else if (strcasecmp(color,"LIGHTGREEN") == 0)
```

```
      col = LIGHTGREEN;
   else if (strcasecmp(color,"YELLOW") == 0)
      col = YELLOW;
   else if (strcasecmp(color,"LIGHTBLUE") == 0)
      col = LIGHTBLUE;
   else if (strcasecmp(color,"LIGHTMAGENTA") == 0)
      col = LIGHTMAGENTA;
   else if (strcasecmp(color,"LIGHTCYAN") == 0)
      col = LIGHTCYAN;
   else if (strcasecmp(color,"WHITE") == 0)
      col = WHITE;
   else
   {
      CONFENTRY* entry = ConfigGetEntry(cfg, NULL,
                                        option, NULL);
      if (entry)
      {
         ErrorOut(NULL,
                  "invalid color definition",
                  "",
                  entry->LineNo,
                  FALSE);
      }
      col = BLACK;
   }
   return col;
}
```

Hier wird nun die Option aus der Datei ausgelesen und anhand von String-Vergleichen in eine Konstante umgewandelt. Die gefundene Farbkonstante wird als Rückgabewert zurückgegeben. Falls ein unsinniger Wert in der Konfigurationsdatei steht, erfolgt eine Fehlermeldung über die Funktion ”ErrorOut”.

Um ein vollständiges Bild zu erhalten sei auf die Lektüre der Datei ”main.c” verwiesen, die in jedem libCUI-Programm zu finden sein sollte.

9. Programmierstil

9.1. Allgemeines

Ein einheitlicher Programmierstil erhöht die Lesbarkeit von Programmen und fördert damit die Transparenz und die Wiederverwendbarkeit von Modulen. Aus diesem Grund werden für libCUI Programme ein paar Hinweise gegeben, wie der Quelltext strukturiert sein sollte.

Diese Hinweise sind weder vollständig noch erheben sie den Anspruch darauf, der Weisheit letzter Schluss zu sein. Vielmehr ist einiges davon Geschmacksache und es bleibt dem Programmierer überlassen, ob er sich daran hält oder nicht.

In dem Augenblick allerdings, in dem Code zur Bibliothek hinzugefügt werden soll (die Mithilfe daran ist jederzeit sehr willkommen), sind diese Hinweise als Richtlinien zu verstehen auf deren Einhaltung bestanden wird. Denn, um die Pflege der hinzugefügten Komponenten zu gewährleisten, ist es erforderlich, dass der Code nicht nur für den ursprünglichen Autor verständlich ist.

Im übrigen ist diese Aufzählung keineswegs vollständig. Falls jemand weiter Hinweise zur Verbesserung der Code-Qualität liefern kann, werden seine Anmerkungen gerne angenommen und ggf. hier hinzugefügt.

9.2. Formatierung

9.2.1. Einrückungen

Einrückungen werden mit dem Tabulatorzeichen und nicht mit Leerzeichen vorgenommen. Der Grund liegt darin, dass viele Editoren und Entwicklungsumgebungen den Programmierer bei der Formatierung des Quelltextes unterstützen (manchmal sogar eine Formatierung erzwingen) und dabei in der Regel immer Tabulator-Zeichen verwenden.

Wird dann an anderer Stelle mit Leerzeichen gearbeitet, dann fällt das Problem auf den ersten Blick nicht auf. In der Ansicht eines anderen Editors (mit anderen Tabulatoreinstellungen) sieht der Quelltext dann aber wie Kraut und Rüben aus, was zu vermeiden ist, indem konsequent mit Tabulatoren gearbeitet wird.

9.2.2. Programmblöcke

Programmblöcke werden in geschweifte Klammern eingeschlossen, wobei die Schreibweise bevorzugt wird, bei der die Klammern übereinander stehen. Also:

```
if (MyFlag == TRUE)
{
   /* do something */
}
else
{
   /* do something else */
}
```

an Stelle von:

```
if (MyFlag == TRUE) {
   /* do something */
}
else {
   /* do something else */
}
```

9.2.3. Ausdrücke

In Ausdrücken werden Operatoren, Zahlen und Variablen durch Leerzeichen voneinander getrennt. Das macht den Quelltext auch in Editoren lesbar, die nicht über Syntax- Hervorhebung verfügen. Also:

```
result = 3 * (x / 4);
```

an Stelle von:

```
result = 3*(x/4);
```

9.2.4. Funktionsnamen

Funktionsnamen werden innerhalb der libCUI in Pascal-Case angegeben. Dabei werden semantische Einheiten des Namens durch Großbuchstaben hervorgerufen. Sie enthalten keine Unterstriche.

Beispiele: ”WindowClose” oder ”EditNew”.

Zudem wird dem Namen der Name des Moduls in dem die Funktion implementiert wurde vorangestellt. Die Funktion ”EditNew” findet sich beispielsweise im Programmmodul (Source-Datei) ”edit.c” wider. Bei sehr langen Modulnamen kann das natürlich entsprechend gekürzt werden.

Funktionen, die nicht an andere Module exportiert werden (lokale Funktionen des Moduls) werden mit dem Schlüsselwort ”static” versehen. Das vermeidet Namenskonflikte, erhöht die Kapselung der Funktion in einem Modul. Zudem kann schon der Compiler auf nicht verwendeten Code hinweisen.

9.2.5. Variablen

Globale Variablen werden in Großbuchstaben geschrieben, wobei ebenfalls die Pascal-Case Notation zum Einsatz kommt. Dabei werden semantische Einheiten des Namens durch Großbuchstaben hervorgerufen. Auch sie enthalten keine Unterstriche.

Lokale Variablen und Funktionsparameter dagegen werden in Kleinbuchstaben geschrieben. Optional kann hier der Unterstrich zur Hervorhebung der Lesbarkeit verwendet werden.

Globale Variablen, die nur innerhalb des Moduls sichtbar sind (Regelfall), erhalten zudem das Schlüsselwort ”static”.

Beispiel:

```
static int MyFlag;

void SwitchItOn(int flag_state)
{
   MyFlag = flag_state;
}
```

9.3. Struktur eines Moduls

Jede Fensterklasse sollte in einer eigenen Source-Datei implementiert werden. Damit erhält man ein Modul pro Fensterklasse.

Innerhalb des Moduls ergibt sich dann der folgende Aufbau:

1. Einbindung von Header-Dateien und Definitionen 2. Typdefinition der Instanzdaten 3. Prototypen lokaler Funktionen 4. Globale Variablen 5. Hook-Funktionen 6. Funktion zum Anlegen der Fensterklasse 7. Zugriffsfunktionen auf die Fensterinstanz 8. Lokale Hilfsfunktionen

Bei Dialogen und anderen Fenstern, die Daten mit anderen Modulen austauschen müssen, kann (und muss) die Typdefinietion der Instanzdaten in der Header-Datei des Moduls erfolgen.

A. Shell Script API

In diesem Kapitel ist die API dargestellt, wie sie für Shell-Script Programmierer zur Verfügung steht. Um die Hintergründe zu verstehen und effizienter arbeiten zu können, sollte man auch das Kapitel ”Scripting” gelesen haben.

Damit die hier beschriebenen Funktionen und Konstanten im Script bekannt sind, sollte die API an oberster Stelle per Source-Include eingebunden werden. Unter eisfair existiert dazu die Datei ”cuilib” die unter ”/var/install/include” abgelegt ist. Einige Programme die eine Script- Erweiterung haben, können zusätzliche include-Dateien anbieten (z.B. die Datei ”ecelib” des eisfair Konfigurationseditors), die unbedingt hinter der allgemeinen API eingebunden werden sollten.

```
#!/bin/sh

. /var/install/include/cuilib      # Standard API
. /var/install/include/ecelib      # optional weitere API

exec_dialog()
{
...
}

cui_init
cui_run
```

A.1. Konstanten

A.1.1. Tastatur-, Farb- und Window-Stil- Konstanten

Key constants

KEY_ENTER	10
KEY_F1	265
KEY_F2	266
KEY_F3	267
KEY_F4	268
KEY_F5	269
KEY_F6	270
KEY_F7	271
KEY_F8	272
KEY_F9	273
KEY_F10	274

Color constants

COLOR_BLACK	0
COLOR_RED	1
COLOR_GREEN	2
COLOR_BROWN	3
COLOR_BLUE	4
COLOR_MAGENTA	5
COLOR_CYAN	6
COLOR_LIGHTGRAY	7
COLOR_DARKGRAY	8
COLOR_LIGHTRED	9
COLOR_LIGHTGREEN	10
COLOR_YELLOW	11
COLOR_LIGHTBLUE	12
COLOR_LIGHTMAGENTA	13
COLOR_LIGHTCYAN	14
COLOR_WHITE	15

Window Styles

CWS_NONE	'0'
CWS_BORDER	'1'
CWS_CAPTION	'2'
CWS_MINIMIZEBOX	'4'
CWS_MAXIMIZEBOX	'8'
CWS_CLOSEBOX	'16'
CWS_SYSMENU	'32'
CWS_RESIZE	'64'
CWS_HIDDEN	'256'
CWS_DISABLED	'512'
CWS_TABSTOP	'1024'
CWS_CENTERED	'2048'
CWS_POPUP	'4096'
CWS_MAXIMIZED	'8192'
CWS_MINIMIZED	'16384'
CWS_STATUSBAR	'32768'
CWS_DEFOK	'65536'
CWS_DEFCANCEL	'131072'
EF_PASSWORD	'16777216'
MF_AUTOWORDWRAP	'16777216'
LB_SORTED	'16777216'
LB_DESCENDING	'33554432'

A.1.2. Window Hook Konstanten

Window Hook Konstanten

HOOK_CREATE	'0'
HOOK_DESTROY	'1'
HOOK_CANCLOSE	'2'
HOOK_INIT	'3'
HOOK_PAINT	'4'
HOOK_NCPAINT	'5'
HOOK_SIZE	'6'
HOOK_SETFOCUS	'7'
HOOK_KILLFOCUS	'8'
HOOK_ACTIVATE	'9'
HOOK_DEACTIVATE	'10'
HOOK_KEY	'11'
HOOK_MMOVE	'12'
HOOK_MBUTTON	'13'
HOOK_VSCROLL	'14'
HOOK_HSCROLL	'15'
HOOK_TIMER	'16'

A.1.3. Control Callback Konstanten

Edit hook types

EDIT_SETFOCUS	'0'
EDIT_KILLFOCUS	'1'
EDIT_PREKEY	'2'
EDIT_POSTKEY	'3'
EDIT_CHANGED	'4'

Label hook types

LABEL_SETFOCUS	'0'
LABEL_KILLFOCUS	'1'

Button hook types

BUTTON_SETFOCUS	'0'
BUTTON_KILLFOCUS	'1'
BUTTON_PREKEY	'2'
BUTTON_POSTKEY	'3'
BUTTON_CLICKED	'4'

Radio hook types

RADIO_SETFOCUS	'0'
RADIO_KILLFOCUS	'1'
RADIO_PREKEY	'2'
RADIO_POSTKEY	'3'
RADIO_CLICKED	'4'

Checkbox hook types

CHECKBOX_SETFOCUS	'0'
CHECKBOX_KILLFOCUS	'1'
CHECKBOX_PREKEY	'2'
CHECKBOX_POSTKEY	'3'
CHECKBOX_CLICKED	'4'

Listbox hook types

LISTBOX_SETFOCUS	'0'
LISTBOX_KILLFOCUS	'1'
LISTBOX_PREKEY	'2'
LISTBOX_POSTKEY	'3'
LISTBOX_CHANGED	'4'
LISTBOX_CHANGING	'5'
LISTBOX_CLICKED	'6'

Combobox hook types

COMBOBOX_SETFOCUS	'0'
COMBOBOX_KILLFOCUS	'1'
COMBOBOX_PREKEY	'2'
COMBOBOX_POSTKEY	'3'
COMBOBOX_CHANGED	'4'
COMBOBOX_CHANGING	'5'

Textview hook types

TEXTVIEW_SETFOCUS	'0'
TEXTVIEW_KILLFOCUS	'1'
TEXTVIEW_PREKEY	'2'
TEXTVIEW_POSTKEY	'3'

Listview hook types

LISTVIEW_SETFOCUS	'0'
LISTVIEW_KILLFOCUS	'1'
LISTVIEW_PREKEY	'2'
LISTVIEW_POSTKEY	'3'
LISTVIEW_CHANGED	'4'
LISTVIEW_CHANGING	'5'
LISTVIEW_CLICKED	'6'

Memo hook types

MEMO_SETFOCUS	'0'
MEMO_KILLFOCUS	'1'
MEMO_PREKEY	'2'
MEMO_POSTKEY	'3'
MEMO_CHANGED	'4'

Terminal hook types

TERMINAL_SETFOCUS	'0'
TERMINAL_KILLFOCUS	'1'
TERMINAL_PREKEY	'2'
TERMINAL_POSTKEY	'3'
TERMINAL_EXIT	'4'

Menu hook types

MENU_SETFOCUS	'0'
MENU_KILLFOCUS	'1'
MENU_PREKEY	'2'
MENU_POSTKEY	'3'
MENU_CHANGED	'4'
MENU_CHANGING	'5'
MENU_CLICKED	'6'
MENU_ESCAPE	'7'

A.1.4. MessageBox- und Dialogkonstanten

MessageBox Styles

MB_NORMAL	'0'
MB_INFO	'16777216'
MB_ERROR	'33554432'
MB_OK	'0'
MB_OKCANCEL	'67108864'
MB_YESNO	'134217728'
MB_YESNOCANCEL	'268435456'
MB_RETRYCANCEL	'536870912'
MB_DEFBUTTON1	'1073741824'
MB_DEFBUTTON2	'2147483648'

Dialog result codes

IDOK	'1'
IDCANCEL	'2'
IDYES	'3'
IDNO	'4'
IDRETRY	'5'

Scroll bar control codes

SB_LINEDOWN	'1'
SB_LINEUP	'2'
SB_PAGEDOWN	'3'
SB_PAGEUP	'4'
SB_THUMBTRACK	'5'

A.2. General Window API

Funktion	Beschreibung
`cui_message()`	MessageBox.
`cui_window_new()`	Create new window instance.
`cui_window_create()`	Create window (and make it visible).
`cui_window_destroy()`	Destroy window (and it's data).
`cui_window_quit()`	Quit application.
`cui_window_modal()`	Execute as modal dialog.
`cui_window_close()`	Close a modal dialog.
`cui_window_sethook()`	Assign a window hook.
`cui_window_getctrl()`	Get a control window.
`cui_getdesktop()`	Get desktop window.
`cui_window_move()`	Move and/or resize a window.
`cui_getwindowrect()`	Get a window's rectangle.
`cui_getclientrect()`	Get a window's client rectangle.
`cui_settimer()`	Start a window timer.
`cui_killtimer()`	Stop a window timer.
`cui_addcolors()`	Add/Modify a color profile.
`cui_hascolors()`	Check if color profile has been defined.
`cui_window_setcolors()`	Assign a color profile to a window.
`cui_window_settext()`	Set window text.
`cui_window_setltext()`	Set window left text.
`cui_window_setrtext()`	Set window right text.
`cui_window_setltext()`	Set window left text.
`cui_window_setstatustext()`	Set window left text.
`cui_window_setltext()`	Set window status text.
`cui_window_setlstatustext()`	Set window left status text.
`cui_window_setrstatustext()`	Set window right status text.
`cui_window_totop()`	Bring window to top.
`cui_window_maximize()`	Maximize window (and reverse).
`cui_window_minimize()`	Minimize window (and reverse).
`cui_window_hide()`	Hide window (and reverse).
`cui_window_enable()`	Enable window (and reverse).
`cui_window_invalidate()`	Invalidate window and force redrawing.
`cui_update()`	Update window stack.
`cui_leave_curses()`	Leave curses mode.
`cui_resume_curses()`	Resume curses mode.
`cui_shell_execute()`	Execute shell script.

A.2.1. cui_message()

Expects:	$1 <−− Parent window	: Window Handle
	$2 <−− Message	: String
	$3 <– Title	: String
	$4 <– Style	: Integer (use $MB_XXX constants)
Returns:	$p2 –> Result code	: Integer

A.2.2. cui_window_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– X	: Integer
	$3 <– Y	: Integer
	$4 <– W	: Integer
	$5 <– H	: Integer
	$6 <– Style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.2.3. cui_message()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Message	: String
	$3 <– Title	: String
	$4 <– Style	: Integer (use $MB_XXX constants)
Returns:	$p2 –> Result code	: Integer

A.2.4. cui_window_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– X	: Integer
	$3 <– Y	: Integer
	$4 <– W	: Integer
	$5 <– H	: Integer
	$6 <– Style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.2.5. cui_window_create()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	

A.2.6. cui_window_destroy()

Expects: $1 <– Window : Window Handle
Returns: <nothing>

A.2.7. cui_window_quit()

Expects: $1 <– Exit code : Integer
Returns: <nothing>

A.2.8. cui_window_modal()

Expects: $1 <– Window : Window Handle
Returns: $p2 –> Result code : Integer

A.2.9. cui_window_close()

Expects: $1 <– Window : Window Handle
$2 <– Result code : Integer
Returns: $p2 –> "1ßuccess : ”0” no success (refused to close)

A.2.10. cui_window_sethook()

Expects: $1 <– Window : Window Handle
$2 <– Hook : Integer (use $HOOK_XXX constants)
$3 <– Proc : String (name of shell function)
Returns: <nothing>

A.2.11. cui_window_getctrl()

Expects: $1 <– Window : Window Handle
$2 <– Ctrl ID : Integer
Returns: $p2 –> Ctrl : Window Handle or ”0” if not found

A.2.12. cui_getdesktop()

Expects: <nothing>
Returns: $p2 –> Desktop : Window Handle

A.2.13. cui_window_move()

Expects:	$1 <– Window	: Window Handle
	$2 <– X	: Integer
	$3 <– Y	: Integer
	$4 <– W	: Integer
	$5 <– H	: Integer
Returns:	<nothing>	

A.2.14. cui_getwindowrect()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> X	: Integer
	$p3 –> Y	: Integer
	$p4 –> W	: Integer
	$p5 –> H	: Integer

A.2.15. cui_getclientrect()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> X	: Integer
	$p3 –> Y	: Integer
	$p4 –> W	: Integer
	$p5 –> H	: Integer

A.2.16. cui_settimer()

Expects:	$1 <– Window	: Window Handle
	$2 <– TimerId	: Integer
	$3 <– Time	: Integer (Milliseconds 100ms resolution)
Returns:	<nothing>	

A.2.17. cui_killtimer()

Expects:	$1 <– Window	: Window Handle
	$2 <– TimerId	: Integer
Returns:	<nothing>	

A.2.18. cui_addcolors()

Expects:	$1 <– Name	: Window Handle
	$2 <– WndColor	: Integer (COLOR_XXX)
	$3 <– WndSelColor	: Integer (COLOR_XXX)
	$4 <– WndTxtColor	: Integer (COLOR_XXX)
	$5 <– SelTxtColor	: Integer (COLOR_XXX)
	$6 <– InactTxtColor	: Integer (COLOR_XXX)
	$7 <– HilightColor	: Integer (COLOR_XXX)
	$8 <– TitleTxtColor	: Integer (COLOR_XXX)
	$9 <– TitleBkgndColor	: Integer (COLOR_XXX)
	$10 <– StatusTxtColor	: Integer (COLOR_XXX)
	$11 <– StatusBkgndColor	: Integer (COLOR_XXX)
	$12 <– BorderColor	: Integer (COLOR_XXX)
Returns:	<nothing>	

A.2.19. cui_hascolors()

Expects:	$1 <– Name	: Name of profile
Returns:	$p2 –> Result	: Integer (0 = not defined, 1 = defined)

A.2.20. cui_window_setcolors()

Expects:	$1 <– Window	: Window Handle
	$2 <– Name	: Name of profile
Returns:	<nothing>	

A.2.21. cui_window_settext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: Window text
Returns:	<nothing>	

A.2.22. cui_window_setltext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: Window text
Returns:	<nothing>	

A.2.23. cui_window_setrtext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: Window text
Returns:	<nothing>	

A.2.24. cui_window_setstatustext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: Window text
Returns:	<nothing>	

A.2.25. cui_window_setlstatustext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: Window text
Returns:	<nothing>	

A.2.26. cui_window_setrstatustext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: Window text
Returns:	<nothing>	

A.2.27. cui_window_totop()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	

A.2.28. cui_window_maximize()

Expects:	$1 <– Window	: Window Handle
	$2 <– State	: 1 = maximize, 0 = normal
Returns:	<nothing>	

A.2.29. cui_window_minimize()

Expects:	$1 <– Window	: Window Handle
	$2 <– State	: 1 = minimized, 0 = normal
Returns:	<nothing>	

A.2.30. cui_window_hide()

Expects: $1 <– Window : Window Handle
$2 <– State : 1 = hidden, 0 = normal
Returns: <nothing>

A.2.31. cui_window_enable()

Expects: $1 <– Window : Window Handle
$2 <– State : 1 = enabled (default), 0 = disabled
Returns: <nothing>

A.2.32. cui_window_invalidate()

Expects: $1 <– Window : Window Handle
Returns: <nothing>

A.2.33. cui_update()

Expects: <nothing>
Returns: <nothing>

A.2.34. cui_leave_curses()

Expects: <nothing>
Returns: <nothing>

A.2.35. cui_resume_curses()

Expects: <nothing>
Returns: <nothing>

A.2.36. cui_shell_execute()

Expects: $1 <– Shell command : String
Returns: <nothing>

A.3. Edit Control API

Funktion	Beschreibung
`cui_edit_new()`	Make new edit.
`cui_edit_callback()`	Assign a edit callback.
`cui_edit_settext()`	Set edit text.
`cui_edit_gettext()`	Get edit text.

A.3.1. cui_edit_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– Len	: Integer
	$8 <– ID	: Integer
	$9 <– Set style	: Integer (use $CWS_XXX constants)
	$10 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.3.2. cui_edit_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.3.3. cui_edit_settext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: String
Returns:	<nothing>	

A.3.4. cui_edit_gettext()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Text	: String

A.4. Label Control API

Funktion	Beschreibung
`cui_label_new()`	Make new label.
`cui_label_callback()`	Assign a label callback.

A.4.1. cui_label_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.4.2. cui_label_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.5. Button Control API

Funktion	Beschreibung
`cui_button_new()`	Make new button.
`cui_button_callback()`	Assign a button callback.

A.5.1. cui_button_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.5.2. cui_button_callback()

Expects:	$1 <– Window : Window Handle
	$2 <– Hook : Integer (use $HOOK_XXX constants)
	$3 <– Target : Window Handle
	$4 <– Proc : String (name of shell function)
Returns:	<nothing>

A.6. Groupbox Control API

Funktion	Beschreibung
`cui_groupbox_new()`	Make new groupbox.

A.6.1. cui_groupbox_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– Set style	: Integer (use $CWS_XXX constants)
	$8 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.7. Radio Control API

Funktion	Beschreibung
`cui_radio_new()`	Make new radio control.
`cui_radio_callback()`	Assign a radio callback.
`cui_radio_setcheck()`	Set radio check status.
`cui_radio_getcheck()`	Get radio check status.

A.7.1. cui_radio_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.7.2. cui_radio_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.7.3. cui_radio_setcheck()

Expects:	$1 <– Window	: Window Handle
	$2 <– Value	: Integer (0 or 1)
Returns:	<nothing>	

A.7.4. cui_radio_getcheck()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Value	: Integer (0 or 1)

A.8. Checkbox Control API

Funktion	Beschreibung
`cui_checkbox_new()`	Make new checkbox control.
`cui_checkbox_callback()`	Assign a checkbox callback.
`cui_checkbox_setcheck()`	Set checkbox check status.
`cui_checkbox_getcheck()`	Get checkbox check status.

A.8.1. cui_checkbox_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.8.2. cui_checkbox_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.8.3. cui_checkbox_setcheck()

Expects:	$1 <– Window	: Window Handle
	$2 <– Value	: Integer (0 or 1)
Returns:	<nothing>	

A.8.4. cui_checkbox_getcheck()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Value	: Integer (0 or 1)

A.9. Listbox Control API

Funktion	Beschreibung
`cui_listbox_new()`	Make new listbox control.
`cui_listbox_callback()`	Assign a listbox callback.
`cui_listbox_add()`	Add string to listbox.
`cui_listbox_delete()`	Delete string from listbox.
`cui_listbox_get()`	Get string from listbox.
`cui_listbox_adddata()`	Assign data to listbox item.
`cui_listbox_getdata()`	Read data from listbox item.
`cui_listbox_setsel()`	Set current selection.
`cui_listbox_getsel()`	Get current selection.
`cui_listbox_clear()`	Clear listbox content.
`cui_listbox_getcount()`	Get number of items in listbox.
`cui_listbox_select()`	Select string in listbox.

A.9.1. cui_listbox_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.9.2. cui_listbox_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.9.3. cui_listbox_add()

Expects:	$1 <– Window	: Window Handle
	$2 <– Value	: String
Returns:	$p2 –> Index	: Integer

A.9.4. cui_listbox_delete()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	<nothing>	

A.9.5. cui_listbox_get()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	$p2 –> Value	: String

A.9.6. cui_listbox_adddata()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
	$3 <– Data	: Integer
Returns:	<nothing>	

A.9.7. cui_listbox_getdata()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	$p2 –> Data	: Integer

A.9.8. cui_listbox_setsel()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	<nothing>	

A.9.9. cui_listbox_getsel()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Index	: Integer

A.9.10. cui_listbox_clear()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	

A.9.11. cui_listbox_getcount()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> NumItems	: Integer

A.9.12. cui_listbox_select()

Expects:	$1 <– Window	: Window Handle
	$2 <– String	: String to select
Returns:	$p2 –> Index	: Integer

A.10. Combobox Control API

Funktion	Beschreibung
`cui_combobox_new()`	Make new combobox control.
`cui_combobox_callback()`	Assign a combobox callback.
`cui_combobox_add()`	Add string to combobox.
`cui_combobox_delete()`	Delete string from combobox.
`cui_combobox_get()`	Get string from combobox.
`cui_combobox_adddata()`	Assign data to combobox item.
`cui_combobox_getdata()`	Read data from combobox item.
`cui_combobox_setsel()`	Set current selection.
`cui_combobox_getsel()`	Get current selection.
`cui_combobox_clear()`	Clear combobox content.
`cui_combobox_getcount()`	Get number of items in combobox.
`cui_combobox_select()`	Select string in the combobox.

A.10.1. cui_combobox_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– X	: Integer
	$3 <– Y	: Integer
	$4 <– W	: Integer
	$5 <– H	: Integer
	$6 <– ID	: Integer
	$7 <– Set style	: Integer (use $CWS_XXX constants)
	$8 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.10.2. cui_combobox_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.10.3. cui_combobox_add()

Expects:	$1 <– Window	: Window Handle
	$2 <– Value	: String
Returns:	$p2 –> Index	: Integer

A.10.4. cui_combobox_delete()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	<nothing>	

A.10.5. cui_combobox_get()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	$p2 –> Value	: String

A.10.6. cui_combobox_adddata()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
	$3 <– Data	: Integer
Returns:	<nothing>	

A.10.7. cui_combobox_getdata()

Expects:	$1 <– Window	: Window Handle
	$2 <– Index	: Integer
Returns:	$p2 –> Data	: Integer

A.10.8. cui_combobox_setsel()

Expects: $1 <– Window : Window Handle
$2 <– Index : Integer
Returns: <nothing>

A.10.9. cui_combobox_getsel()

Expects: $1 <– Window : Window Handle
Returns: $p2 –> Index : Integer

A.10.10. cui_combobox_clear()

Expects: $1 <– Window : Window Handle
Returns: <nothing>

A.10.11. cui_combobox_getcount()

Expects: $1 <– Window : Window Handle
Returns: $p2 –> NumItems : Integer

A.10.12. cui_combobox_select()

Expects: $1 <– Window : Window Handle
$2 <– String : String to select
Returns: $p2 –> Index : Integer

A.11. ProgressBar Control API

Funktion	Beschreibung
`cui_progress_new()`	Make new progress bar control.
`cui_progress_setrange()`	Set bar range.
`cui_progress_setpos()`	Set bar pos.
`cui_progress_getrange()`	Get bar range.
`cui_progress_getpos()`	Get bar pos.

A.11.1. cui_progress_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.11.2. cui_progress_setrange()

Expects:	$1 <– Window	: Window Handle
	$2 <– Range	: Integer
Returns:	<nothing>	

A.11.3. cui_progress_setpos()

Expects:	$1 <– Window	: Window Handle
	$2 <– Pos	: Integer
Returns:	<nothing>	

A.11.4. cui_progress_getrange()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Range	: Integer

A.11.5. cui_progress_getpos()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Pos	: Integer

A.12. Textview Control API

Funktion	Beschreibung
`cui_textview_new()`	Make new textview control.
`cui_textview_callback()`	Assign a textview callback.
`cui_textview_wordwrap()`	Enable word wrap.
`cui_textview_add()`	Add string to textview.
`cui_textview_clear()`	Clear text view.
`cui_textview_read()`	Read a file.
`cui_textview_search()`	Search within textview.

A.12.1. cui_textview_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.12.2. cui_textview_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.12.3. cui_textview_wordwrap()

Expects:	$1 <– Window	: Window Handle
	$2 <– Enable?	: Integer (0 or 1)
Returns:	<nothing>	

A.12.4. cui_textview_add()

Expects:	$1 <– Window	: Window Handle
	$2 <– Value	: String
	$3 <– Update	: Flag (1 = update view, 0 = no update)
Returns:	<nothing>	

A.12.5. cui_textview_clear()

Expects:	$1 <– Window	: Window Handle
	$2 <– Value	: String
Returns:	<nothing>	

A.12.6. cui_textview_read()

Expects:	$1 <– Window	: Window Handle
	$2 <– Filename	: String
Returns:	$p2 –> Success	: Integer (0 = failed, 1 = success)

A.12.7. cui_textview_search()

Expects:	$1 <– Window	: Window Handle
	$2 <– SearchStr	: String
	$3 <– WholeWord	: Integer (0 or 1)
	$4 <– CaseSense	: Integer (0 or 1)
	$5 <– Down	: Integer (0 or 1)
Returns:	$p2 –> Success	: Integer (0 = failed, 1 = success)

A.13. Listview Control API

Funktion	Beschreibung
`cui_listview_new()`	Make new listview control.
`cui_listview_callback()`	Assign a listview callback.
`cui_listview_setcoltext()`	Set listview column titel.
`cui_listview_clear()`	Clear listview data.
`cui_listview_add()`	Add a new (empty) record to the list view (must be populated with data using cui_listview_settext)
`cui_listview_settext()`	Assign text data to listview record.
`cui_listview_gettext()`	Get text data assigned listview record.
`cui_listview_setdata()`	Assign integer data to listview record.
`cui_listview_getdata()`	Get integer data from listview record.
`cui_listview_setsel()`	Set current selection in listview control.
`cui_listview_getsel()`	Get current selection of listview control.
`cui_listview_getcount()`	Get number of records in listview control.
`cui_listview_update()`	Update listview visual display after modifying data.

A.13.1. cui_listview_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– Num columns	: Integer
	$8 <– ID	: Integer
	$9 <– Set style	: Integer (use $CWS_XXX constants)
	$10 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.13.2. cui_listview_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.13.3. cui_listview_setcoltext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Col-No	: Integer
	$3 <– Text	: String
Returns:	<nothing>	

A.13.4. cui_listview_clear()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	

A.13.5. cui_listview_add()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Index of new record	: Integer

A.13.6. cui_listview_settext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Rec. index	: Integer (returned by cui_listview_add)
	$3 <– Col. index	: Integer (0 | 1 | ... | cols - 1)
	$4 <– Text	: String
Returns:	<nothing>	

A.13.7. cui_listview_gettext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Rec. index	: Integer (returned by cui_listview_add)
	$3 <– Col. index	: Integer (0 | 1 | ... | cols - 1)
Returns:	$p2 –> Text	: String

A.13.8. cui_listview_setdata()

Expects:	$1 <– Window	: Window Handle
	$2 <– Rec. index	: Integer (returned by cui_listview_add)
	$3 <– Data	: Integer
Returns:	<nothing>	

A.13.9. cui_listview_getdata()

Expects:	$1 <– Window	: Window Handle
	$2 <– Rec. index	: Integer (returned by cui_listview_add)
Returns:	$p2 –> Data	: Integer

A.13.10. cui_listview_setsel()

Expects:	$1 <– Window	: Window Handle
	$2 <– Rec. index	: Integer (returned by cui_listview_add)
Returns:	<nothing>	

A.13.11. cui_listview_getsel()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Sel. index	: Integer (-1 if not selected)

A.13.12. cui_listview_getcount()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Count	: Integer

A.13.13. cui_listview_update()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	

A.14. Memo Control API

Funktion	Beschreibung
`cui_memo_new()`	Make new memo.
`cui_memo_callback()`	Assign a memo callback.
`cui_memo_settext()`	Set memo text.
`cui_memo_gettext()`	Get memo text.

A.14.1. cui_memo_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Window title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.14.2. cui_memo_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.14.3. cui_memo_settext()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: String
Returns:	<nothing>	

A.14.4. cui_memo_gettext()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> Text	: String

A.14.5. cui_memo_setwrapcolumns()

Expects:	$1 <– Window	: Window Handle
	$2 <– Column	: String
Returns:	<nothing>	

A.15. Terminal Window API

Funktion	Beschreibung
`cui_terminal_new()`	Make new terminal control.
`cui_terminal_callback()`	Assign a terminal callback.
`cui_terminal_write()`	Write text into the terminal window.
`cui_terminal_run()`	Run a command within the terminal window.
`cui_terminal_pipe()`	Pipe text data to the running co process.

A.15.1. cui_terminal_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.15.2. cui_terminal_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.15.3. cui_terminal_write()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: String
	$3 <– DoUpdate	: Integer
Returns:	<nothing>	

A.15.4. cui_terminal_run()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: String
Returns:	$p2 –> Success	: Integer (0 or 1 !not the exit code of the process!)

A.15.5. cui_terminal_pipe()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: String
Returns:	<nothing>	

A.16. Menu Window API

Funktion	Beschreibung
`cui_menu_new()`	Make new menu control.
`cui_menu_callback()`	Assign a menu callback.
`cui_menu_additem()`	Add a menu item.
`cui_menu_addseparator()`	Add a menu separator.
`cui_menu_selitem()`	Select menu item.
`cui_menu_getselitem()`	Get selected menu item.
`cui_menu_clear()`	Clear menu.

A.16.1. cui_menu_new()

Expects:	$1 <– Parent Window	: Window Handle
	$2 <– Title	: String
	$3 <– X	: Integer
	$4 <– Y	: Integer
	$5 <– W	: Integer
	$6 <– H	: Integer
	$7 <– ID	: Integer
	$8 <– Set style	: Integer (use $CWS_XXX constants)
	$9 <– Clear style	: Integer (use $CWS_XXX constants)
Returns:	$p2 –> New Window	: Window Handle

A.16.2. cui_menu_callback()

Expects:	$1 <– Window	: Window Handle
	$2 <– Hook	: Integer (use $HOOK_XXX constants)
	$3 <– Target	: Window Handle
	$4 <– Proc	: String (name of shell function)
Returns:	<nothing>	

A.16.3. cui_menu_additem()

Expects:	$1 <– Window	: Window Handle
	$2 <– Text	: String
	$3 <– Id	: Item id
Returns:	<nothing>	

A.16.4. cui_menu_addseparator()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	

A.16.5. cui_menu_selitem()

Expects:	$1 <– Window	: Window Handle
	$2 <– Id	: Integer (item id)
Returns:	<nothing>	

A.16.6. cui_menu_getselitem()

Expects:	$1 <– Window	: Window Handle
Returns:	$p2 –> ItemId	: Integer

A.16.7. cui_menu_clear()

Expects:	$1 <– Window	: Window Handle
Returns:	<nothing>	