Design Resource Wizard

Design

Version 1.0

This manual was produced using *Doc-To-Help*®, by WexTech Systems, Inc.

WexTech Systems, Inc.
310 Madison Avenue, Suite 905
New York, NY 10017
+1 (212) 949-9595

Inhalt

Analyse

Zweck der Applikation

Der **ResourceWizzard** unterstützt die Entwicklung von mehrsprachigen Applikationen in MS Visual C++ . Für diesen Zweck vereinfacht er den Umgang mit Ressourcen, bzw. den Umgang mit den von MS Visual C++ erstellten Ressource Dateien.

Folgende Arbeiten werden für den Entwickler vereinfacht oder automatisiert:

- Generieren von mehrsprachigen Ressource Dateien aus einer einsprachigen.
- Extrahieren einer einsprachigen Ressource Datei aus einer mehrsprachigen.
- Hinzufügen und Entfernen von Ressourcen - Teilen einer Sprache oder mehreren Sprachen.
- Übersetzung von Ressource Texten automatisch (Autotranslate) und manuell. Bei manueller Übersetzung: Vorschläge aus Dictionnary
- Abgleichen der Ressourcen verschiedener Sprachen in Inhalt und Format.
- Validierung der Hotkeys (Menüs, Dialog, etc.)
- Voransicht der übersetzten Ressource (Dialog, etc.)
- Rudimentäre Versionsverwaltung, insbesondere erkennen, was seit der letzten Sitzung geändert hat (Übersetzungslücken, Inhomogenitäten, etc.)

Anforderungen

Die Applikation soll in der Lage sein, eine Ressource Datei einzulesen und in einer internen Struktur abzuspeichern. Diese Struktur soll nun gemäss den oben genannten Möglichkeiten verändert werden können. Danach soll sie (oder Teile davon) wieder in Form einer Ressource Datei abgespeichert werden.

Die vom Benützer durchgeführten Manipulationen können die ganze Struktur oder auch nur Teile davon betreffen. In jedem Fall aber werden Texte ersetzt werden müssen. Die dazu benötigten Texte sind in einer Datenbank gespeichert.

Die Bedienung des ResourceWizzard soll für den Benutzer einfach und klar verständlich sein. Die Bedienung geschieht ausschliesslich über die GUI der Applikation. Der Aufbau dieser GUI und die Darstellung der darin gezeigten Informationen sollte darum einfach und übersichtlich sein.

Interne Struktur der Resourcen

Intern werden die Ressourcen in Form einer Baumstruktur abgelegt. Diese Baumstruktur muss einfach und sicher verändert werden können. Insbesondere werden Methoden gebraucht, die Teilbäume kopieren, oder löschen können. Ebenfalls benötigt werden Methoden, die Texte aus dieser Baumstruktur extrahieren oder ersetzen können. Beim Einlesen einer Ressource Datei werden **sämtliche** Informationen in den Baum übertragen. Es gibt also keinen Informationsverlust!

Parsing

Das Umwandeln von Ressource Dateien in eine Baumstruktur und umgekehrt basiert auf der Syntax von MS Visual C++ Ressource Dateien. (Syntaxdiagramme siehe Anhänge). Andere Ressourcen werden also **nicht** unterstützt. Die benötigten Algorithmen können aus dieser Syntax direkt abgeleitet werden.

Der Parser soll in einem Modul gekapselt werden, damit für spätere Anforderungen (weitere Ressourcenformate) nur in diesem Teil Erweiterungen vorgenommen werden müssen.

Datenbank

Die Datenbank speichert die zur automatischen Übersetzung notwendigen Texte. Sie muss mit derjenigen von WMLS übereinstimmen. Die Beschreibung der Datenbank kann dem Benutzerhandbuch WMLS, Version 1.1 entnommen werden.

Testfälle/Abnahmekriterien

Der ResourceWizard erfüllt die Anfroderungen, falls die nachfolgend beschriebenen Testfälle mit Erfolg verarbeitet werden konnten. Aus diesem Grund werden für die Abnahme ebendiese Testfälle auf das erzielte Resultat angewendet. Als Resultate der Abnahme können die folgenden Fälle eintreten:

- Abnahme erfolgreich
- Abnahme mit Vorbehalten (Der Rückbehalt kann eingefordert werden, die Garantiefirst beginnt. Die Mängel werden aufgelistet und müssen binnen Garantiefirst behoben werden.)
- keine Abnahme. (Die Mängel werden genannt und zur Korrektur angeboten. Danach wird die Abnahme erneut durchgeführt. Die Vertragspartner können gegebenenfalls unter Aushandlung der Konditionen den Vertrag auslösen.)

Testfälle

Als Basistests wird das Programm auf die in den Anforderungen enthaltenen Kriterien ungeordnet getestet. Insbesondere werden neue Sprachen hinzugefügt, die automatische und manuelle Übersetzung geprüft, und das Look-and-Feel bewertet.

Darüber hinaus werden geordnete Testfälle betrachtet, welche in den nachfolgenden Abschnitten beschreiben sind.

Einsprachige Ressource Dateien

Den Tests wird eine vollständige einsprachige Ressource Datei zu Grunde gelegt.

Tests:

1. Einlesen der Datei. Speichern der Datei
   Die Dateien müssen identisch sein
2. Einlesen der Datei. Hinzufügen einer (mehreren) Sprache(n). Speichern der Datei
   Die Datei muss sich in MSVC++ kompilieren lassen
3. Einlesen der Datei. Hinzufügen einer Sprache. Löschung eines Teils der fremdsprachigen Ressource. Speichern der Datei
   Die Datei muss sich in MSVC++ kompilieren lassen

Mehrsprachige Ressource Dateien

Den Tests wird eine beliebige mehrsprachige Ressource Datei zu Grunde gelegt.

Tests:

1. Einlesen der Datei. Speichern der Datei
   Die Dateien müssen identisch sein
2. Ändern eines Textes in einer Ressource im Developer Studio (MSDEV)
   Die Änderung muss erkannt werden. Die Übersetzungen müssen angepasst werden.
3. Ändern des Layouts einer Dialoge Ressource im Developer Studio (MSDEV)
   Die Änderung muss erkannt werden. Die Homogenisierung muss angeboten werden.
4. Einlesen der Datei. Löschung einer Sprache. Speichern der Datei
   Die Datei muss sich in MSVC++ kompilieren lassen
5. Einlesen der Datei. Generierung einer einsprachigen Datei
   Die Datei muss sich in MSVC++ kompilieren lassen

Datenbank

Zu jedem Zeitpunkt der Verarbeitung muss garantiert sein, dass sich die Datenbank in einem konsistenten Zustand befindet. Geprüft wird dies, indem mit WMLS die Datenbank gelesen wird.

Grobdesign

Komponenten

Die Applikation besteht grob gesagt aus folgenden Komponenten:

- Die **Datenbank**, speichert die Texte der verschiedenen Sprachen.
- Der **Datenbankmanager** verwaltet die Datenbank. Nur über ihn kann auf die Datenbank zugegriffen werden.
- Die **GUI** kapselt alle Bildschirmorientierte Funktionalität. Sie löst Aktionen der anderen Komponenten aus.
- Komponenten, welche die Informationen der Ressourcen beinhalten. Die wichtigsten Komponenten sind der **Ressourcenbaum** und die **Texttabelle**.

Im Folgenden werden alle diese Komponenten detailliert vorgestellt. Das Herzstück der Applikation bilden die internen Komponenten, welche die Informationen der Ressourcen darstellen.

Einbettung

Die folgende Grafik gibt eine Übersicht über die ganze Applikation.

Fehler! Keine gültige Verknüpfung.

Ablauf der Applikation

Der Ablauf der Applikation erfolgt grob in folgenden Stufen:

1. Die gewünschte Ressource Datei wird eingelesen. Dabei werden die Texttabelle und der Ressourcenbaum aufgebaut.
2. Die Ressourcen werden abgeändert.
3. Das Resultat wird gespeichert.

Einlesen der Ressourcen

Die gewünschte Datei wird vom Benützer angegeben. Diese wird dann geparst und die Informationen werden im Ressourcenbaum abgespeichert. Alle ressourcespezifische Information wie Koordinaten, IDs etc. wird in dem Baum gespeichert. Die in den Ressourcen enthaltenen Texte werden in der Texttabelle erfasst. Die Einträge in dieser Tabelle sind einmalig. So wird zum Beispiel das "Datei" nur einmal erfasst, auch wenn es an mehreren Stellen auftaucht. Erst wenn für das Wort "Datei" mehrere Übersetzungsvarianten existieren erscheint das Wort in der Tabelle mehrmals.

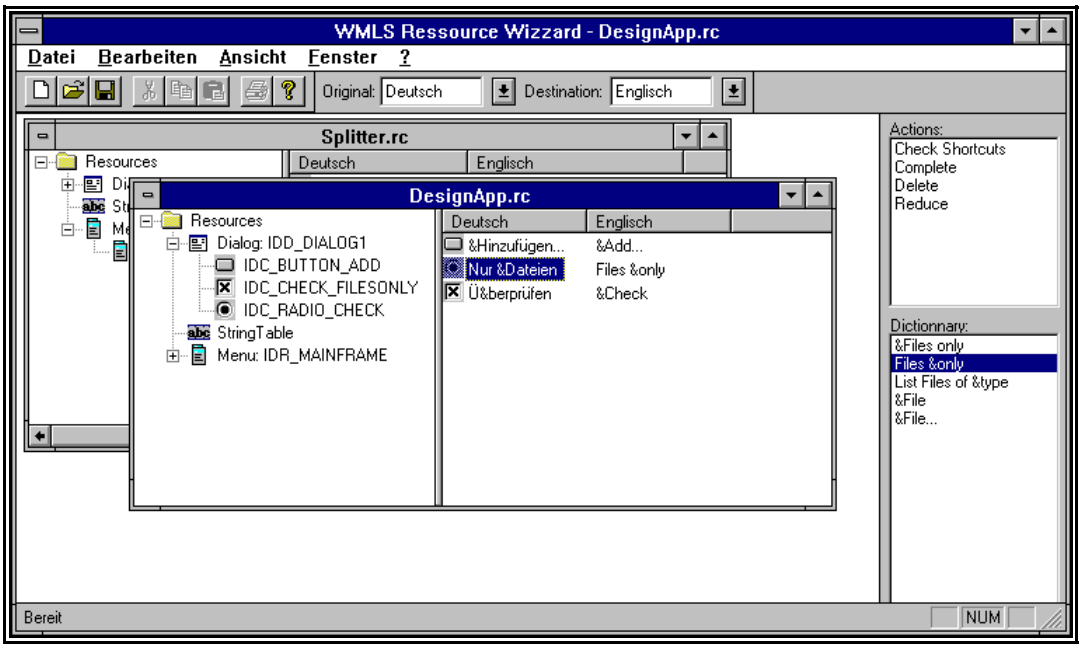
Verändern der Resourcen

Der Benutzer kann nun über die GUI Ressourcen vervollständigen oder löschen oder bestehende Ressourcen übersetzen. Die genauen Möglichkeiten werden weiter unten, im Kapitel über das GUI, beschrieben.

Speichern

Der Benutzer kann nun aus den veränderten bzw. erweiterten Ressourcen neue Ressourcedateien erstellen. Er kann dabei wählen, ob er die ganzen Ressourcen oder nur Teile davon übernehmen möchte. Details darüber ebenfalls im Kapitel über das GUI.

Screendesign



Dieses Screendesign ist keinesfalls definitiv. Es ist lediglich ein Vorschlag!

Hier ein Vorschlag für das GUI:

Beim Start der Applikation muss ein SplashWindow erscheinen.

Die Applikation muss die Datenbankverbindung selbständig auf- und abbauen können.

Die Kontrollelemente Toolbars etc. müssen dockbar sein. Die Einstellungen sollen für die nächste Sitzung gespeichert werden (benutzerabhängig).

Im Menü muss eine Dateigeschichte angeboten werden (z.B. die 4 zuletzt verwendeten Dateien.

Die Applikation muss mehrsprachig sein (mehrsprachiges Resourcefile) und auch andere Zeichensätze unterstützen (UNICODE). WMLS ResourceWizard soll verwendet werden können, um den WMLS ResourceWizard mehrsprachig zu machen!!

Die Applikation muss streng nach den **Richtlinien der Software Entwicklung der TeTrade AG** (Programmierstandards, Namenskonventionen) entwickelt worden sein.

Für jeden Menüpunkt muss eine Entsprechung auf der Toolbar zur Verfügung gestellt werden.

Bildschirmelemente und -bereiche

Die Applikation ist MDI-fähig. Die einzelnen Ressourcefiles werden in den entsprechenden Kindfenstern dargestellt. Das Hauptfenster hat ein Menü, eine Werkzeugleiste für die Standardaktionen, eine Werkzeugleiste für die Spracheinstelllungen und eine ControlBar, welche in Abhängigkeit der aktuellen Selektion im Kindfenster des MDIs die möglichen Aktionen und Übersetzungen anzeigt.

Generelle Eigenschaften der Applikation

Kindfenster

Ein Kindfenster enthält auf der linken Seite den visualisierten Ressourcebaum (ContextView) und auf der rechten Seite die aktuelle Originalsprache und eine der gewählten Zielsprachen (TextView).

Bedienelemente

Der ContextView

Der ContextView illustriert die Struktur der Resourcen, und zwar mit Hilfe eines TreeCtrl. Es ist multiselektierbar. Alle möglichen Operationen wirken sich nur auf die hier selektierten Teile der Resourcen aus.

Die Darstellung des TreeCtrl entspricht im wesentlichen der im Visual C++ (Resource Editor). Allerdings gibt es verschiedene Icons für die verschidenen darin enthaltenen Ressourceelemente. Das Icon für ein bestimmtes Item sieht anders aus, je nach dem in welchem Zustand sich das Item befindet. Es gibt folgende Zustände:

1. **Complete**: Die Struktur des Items ist in der Original- und in der ZielSprache identisch.
2. **Uncomplete**: In der Struktur des Items existieren Elemente für die Originalsprache, die für die Zielsprache nicht existieren.
3. **Deleted**: In der Struktur des Items existieren Elemente für die Zielsprache, die für die Originalsprache nicht existieren.

Bei Beginn ist alles bzw. das Wurzelelement selektiert.

Die Werkzeugleiste für die Sprachwahl

Die Werkzeugleiste für die Sprachwahl enthält zwei ComboBoxen. Eine dient zur Wahl der Originalsprache. Sie wird quasi als Referenzsprache benutzt für die Texte in der zu übersetzenden Zielsprache. Die Zielsprache kann in der anderen ComboBox ausgewählt werden.

Beim öffnen der Ressourcedatei muss die Originalsprache ausgewählt werden. Zur Auswahl stehen alle Sprachen, die **schon in den Ressourcen vorkommen**. Danach kann die Zielsprache gewählt werden. Wiederum stehen alle Sprachen, **schon in den Ressourcen enthalten sind**, zur Verfügung - natürlich ohne die Originalsprache.

Der TextView

Der TextView enthält zwei Spalten. Einzelne Zeilen sind selektier- und editierbar. Es zeigt alle extrahierten Texte, die aus Strukturelementen stammen, welche im Kontextfenster selektiert sind. In der linken Spalte werden die Texte in Originalsprache und in der rechten Spalte in der Zielsprache angezeigt. Zusätzlich ist jeweils am linken Rand beider Spalten ein Icon zu sehen, das anzeigt aus welcher Ressource die Texte stammen. Es gibt dafür eine ganze Zahl von Icons z.B. für Dialoge, Menüs, Stringtables etc. Alle diese Icons gibt es in zwei Varianten- durchgestrichen oder nicht. Je nachdem ob das Resourceelement in der Sprache exisiert, erscheint es durchgestrichen oder nicht. Wenn der Anwender darauf klickt, wechselt das Icon von durchgestrichen zu nicht durchgestrichen oder umgekehrt und die bezeichnete Struktur wird in der spezifizierten Sprache angelegt resp. gelöscht.

Existiert ein Text in einer Sprache nicht, so erscheint das Feld leer. Der Benutzer kann nun das Feld selektieren und den gewünschten Text eingeben oder aber er wählt einen Übersetzungsvorschlag, der ihm aus dem Dictionnary angeboten wird.

Neue erfasste Texte erscheinen **grau** bis die Ressourcedatei gespeichert worden ist.

ActionView in der ControlBar

Der ActionView bietet dem Benutzer alle für eine gewissen Zustand (selektierte Elemente in den Kindfenstern) möglichen Aktionen an. Mögliche Aktionen sind mindestens "Complete", "Reduce" und "Delete". Drückt man auf die Taste "**Complete**" werden sämtliche Strukturen die selektiert sind insofern komplettiert, dass alles was für die Originalsprache existiert auch für die Zielsprache angelegt wird (sofern es nicht schon existiert). Wird die Taste "**Reduce**" gedrückt werden sämtliche Strukturelemente die in der Zielsprache, nicht aber in der Originalsprache existieren, gelöscht. Beim Drücken der Taste "**Delete**" werden sämtliche selektierten Strukturelemente für die Zielsprache gelöscht.

Der Dictionnary

Im DictionnaryView werden die für einen selektierten Originaltext die möglichen Übersetzungen in die aktuell eingestellte Zielsprache vorgeschlagen. Der Dictionnary bezieht sein Wissen (Übersetzungstabelle) aus den Erfahrungen (bisherige Übersetzungen) und aus dem Grundwissen (Standardübersetzungs-tabelle aus der Installation). Die beiden Quellen sollte farblich oder ähnlich unterschieden werden können.

Benutzeraktionen und deren Wirkung

Mausaktionen und Tastaturaktionen

Selektion eines Ressourceelementes im ContextView
    TextView aktualisieren
    ActionView aktualisieren

Selektion eines Textelementes im TextView
    DictionnaryView aktualisieren
    ActionView aktualisieren

Doppelklick auf ein Textelementes im TextView
    Reduce oder Delete auslösen

Doppelklick auf ein Textelementes im DictionnaryView
    Zieltext in der aktuellen Selektion ersetzen.

Doppelklick auf ein Aktion im ActionView
    Aktion auslösen

Klick auf Spaltentitel im TextView
    Alphabetische Sortierung nach der entsprechenden Spalte
    (alternierend: aufsteigend, absteigend, unsortiert)

Rechte Maustaste
    ContextMenü (Popup-Menü) erscheint und bietet sinnvolle
    Aktionen für den Kontext an

Menü Datei

Öffnen...

Der Benutzer kann über das Menu `File/Open...` eine Resourcedatei öffnen. Die Datei wird dann geparst und die ensprechende Baumstruktur und Texttabelle aufgebaut. Die Struktur des Baumes erscheint im Kontextfenster.

Wenn die Ressourcedatei bereits mehrere Sprachen enthält, wird die Originalsprache beim Benutzer erfragt. Diese Information wird als Eigenschaften des Übersetzungsprojektes (Ressourcefile) gespeichert.

Der TextView wird aktualisiert. Wenn die geöffnete Datei keine Zielsprache enthält, wird keine Zielsprache angezeigt. Enthält sie mehr als eine, wird diejenige der letzten Sitzung angezeigt.

Schliessen

Standardverhalten

Speichern... / Speichern unter...

Nach getaner Arbeit kann der Benutzer über das Menu `Datei/Speichern unter ...` die neuen Resourcen (oder Teile davon) wieder in Form eines Resourcefiles ablegen. Wiederum entscheidet die Selktion in dem Kontextfenster was alles ins neue Resourcefile kommt.

Drucken... / Seitenansicht / Druckereinrichtung

Standardverhalten

Beenden

Standardverhalten. Insbesondere Hinweis auf noch nicht gespeicherte Dateien!!!

Menü Bearbeiten

Rückgängig

Standardverhalten. Die letzten 256 Aktionen (sofern sinnvoll und möglich) sollen rückgängig gemacht werden können. Dies betrifft insbesondere die Löschungen und Texteingaben.

Kopieren/Ausschneiden/Einfügen

Standardverhalten

Menü Ansicht

Ressource Baum / Textdatei

Eine Ressourcedatei kann als Ressourcebaum (vgl. Abbildung) oder als Textdatei dargestellt werden. Über diesen Menüpunkt soll die gewünschte Ansicht ausgewählt werden können.

Toolbar

Standardverhalten (Anzeigen und verstecken)

LanguageBar

Standardverhalten (Anzeigen und verstecken)

ControlBar

Standardverhalten (Anzeigen und verstecken)

Statusbar

Standardverhalten (Anzeigen und verstecken)

Menü Extras

Wählt man "**AutoTranslate**", versucht die Applikation mit Hilfe der Datenbank alle fehlenden Texte zu ergänzen. Die so neu hinzukommenden Wörter erscheinen grau, was bedeutet, dass sie noch nicht in den Ressourcen bzw. Texttabelle gespeichert sind.

Diese Speicherung kann mit der Wahl von Datei/Speichern forciert werden. Beim Drücken dieser Taste erscheint eine Dialogbox, über die der Benutzer entscheiden, kann ob die von ihm manuell eingegebenen Texte in der Datenbank gespeichert werden sollen.

In der Datenbank werden möglicherweise mehrere Einträge gefunden. Selektiert man ein Feld das grau ist, werden in der Listbox ("others") alle gefundenen Alternativen angezeigt.

AutoTranslate...

Dialog Viewer...

Dieser Menüpunkt zeigt - sofern im Kindfenster ein Dialog selektiert worden ist - den Dialog in Abhängigkeit der aktuellen Zielspracheinstellung in seiner Erscheinungsform an. Diese Funktionalität wird zur Prüfung der ausreichenden Grösse der Controls genutzt.

Dieser Menüpunkt erlaubt die sprachabhängige Veränderung der Grösse der Control, indem der Dialog im Entwurfsmodus angezeigt wird und so überarbeitet werden kann. Das Verhalten dieses Wizzards ist an das Verhalten des von der Entwicklungsumgebung Visual C++ bekannten Dialogedtitors anzulehnen.

Dieser Menüpunkt veranlasst die Applikation zur Validierung der Schnellwahltasten (z.B. Alt+B für „&Beenden“) für sämtliche Controls eines Dialoges. Es wird auf Doppelbelegungen geachtet. Shortcuts (z.B. Ctrl-O für Datei-Öffnen) können nicht geändert müssen und werden deshalb auch nicht validiert werden.

Durch die Wahl dieses Menüpunktes wird in den Verwaltungsdialog für den Dictionnary verzweigt hier können Übersetzungstexte erfasst, verändert oder gelöscht werden. Zudem können Texte importiert oder exprotiert werden.

Dialog Wizard...

Validate Hotkeys...

Resource Wizard Dictionnary...

Dieser Menüpunkt soll die spätere Anbindung verschiedener Dictionnaries erlauben.

Dictionnary...

Der Optionsdialog soll Benutzerspezifische Anpassungen, wie Wahl der Schriftart und Grösse, Zeichensatz für die Übersetzung in nicht ASCII basierte Sprachen erlauben.

Optionen...

Menü Fenster

Standard MDI-Window Menü

Menü Hilfe

Standard Hilfe-Menü

Detaildesign

Im Folgenden werden alle diese Komponenten detailliert vorgestellt. Das Herzstück der Applikation bilden die internen Komponenten, welche die Informationen der Resourcen darstellen. Sie werden deshalb als erstes besprochen

Die Texttabelle

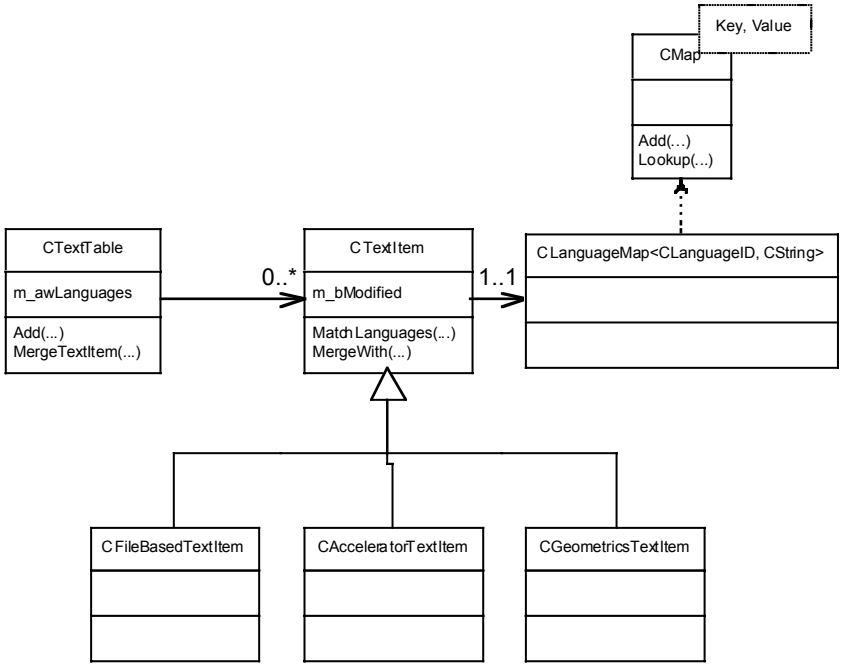
Alle Texte, die in den Resourcen vorkommen (Menutitel, Pushbutton-bezeichnungen etc.) werden in einer Texttabelle verwaltet **und zwar ausschliesslich dort**. Diese Texttabelle ist eine Liste von Textitems. Die Textitems speichern ein zusammengehörendes Set von Wörtern in verschiedenen Sprachen. Die Texttabelle kann man sich ungefähr folgendermassen vorstellen:

Deutsch	Englisch	Französich
Datei	File	Fichier
Datei	Files	Fichier
Dateien	File	Fichier

Grundsätzlich kommt ein Wort nur einmal in der Texttabelle vor. Nur wenn es mehrere Übersetzungen für ein und dasselbe Wort gibt, kann es mehrmals auftreten. Dazu ein Beispiel: Nehmen wir an, die Recource Datei enthalte ein Dialog und ein Menu und beide seien schon in Deutsch und Englisch vorhanden. Nehmen wir nun weiterhin an, das Wort "Datei" kommt in beiden vor. Am einen Ort, zum Beispiel im Menu, ist der entsprechende englische Text "File" und am andern Ort "Files" in diesem Fall wird es zwei Einträge in der Texttabelle geben, da in den gleichen Resourcen "Datei" offensichtlicherweise einmal mit "Files" und eimal mit "File" übersetzt wird. (Dem Beispiel entsprechen Zeile 1 und 2 in der obigen Tabelle.)

Zusätzlich hat jedes Textitem eine Liste mit Referenzen, die diejenigen Stellen des Recourcen Baumes bezeichnen, welche sich auf das Textitem beziehen bzw. es verwenden. Die Absicht davon ist klar: Tritt ein Text an verschiedenen stellen in der Resource - Datei auf, so wird sie trotzdem nur einmal gespeichert.

Klassendiagramm Texttabelle



Die Klasse CTextItem

Die Definition der Klasse **`CTextItem`** sieht nun so aus:

```
class CTextItem {
    // Methods
        CTextItem();
        ~CTextItem();
        CTextItem& CTextItem(const CTextItem);

        BOOL MatchLanguages(CTextItem &compareWith);
        void AddReference(CTreeItem *lpTreeItem);
        CString * GetText(DWORD dwLanguageId);
        void SetText(CString strText, DWORD dwLanguageID);

    // Attributes
        CLanguageMap Texts;
        CPtrList References;
};
```

Der Copyconstructor kopiert ein TextItem. Dabei wird **nur** die LanguageMap kopiert. Die **`References` Liste des neuen Items wird leer initialisiert.**

`MatchLanguages()` vergleicht alle Sprachen der beiden TextItems (this und compareWith). Falls **alle** Sprachen übereinstimmen, wird `TRUE` zurückgegeben, andernfalls `FALSE`. Diese Methode kann auch als Vergleichsoperator `CTextItem::Operator==()` implementiert werden.

`AddReference()` hängt ein neue **`CTreeItem`** - Referenz der `References` - Liste an.

`GetText()` gibt ein Pointer auf den CString zurück, der den Text für die Sprach mit ID `dwLanguageID` enthält. Wird kein Eintrag für diese ID gefunden, wird `NULL` zurückgegeben.

`SetText()` fügt einen neuen Text zur éanguageMap hinzu, und zwar unter der ID `dwLanguageID`. Ist der Text schon vorhanden, wird er überschrieben.

Die Klasse **`CLanguageMap`** dient der Speicherung der Texte in den verschiedenen Sprachen. Dabei werden die Texte jeweils unter einer eindeutigen ID, ein `DWORD`, abgelegt. Die ID wird aus der SprachID und der UnterspachID (z.B. SUB_LANG_GERMAN_SWISS) gebildet. Für die Generierung der Ids existiert in der MFC-Bibliothek ein Makro:

```
dwLanguageID = MAKELONG( wSubLangID, wLangID);
```

`CLanguageMap` ist eine Template-Realisierung der MFC-Klasse CMap. Eigentlich werden dafür vier und nicht nur zwei Typen benötigt, allerdings hat eine Realisierung immer so

CMap<Key, Key&, Value, Value&>

aus. `CLanguageMap` mappt somit einfach Sprach - IDs nach Strings.

Die Derivate von CTextItem dienen dazu, andere Aspekte der Resourcen sprachabhängig zu erfassen. Beispielsweise bildet CGeometricsTextItem Sprach - IDs auf Rechteckwerte ab. Auch hier wird eine CMap - Realisierung benutzt (von CLanguageID auf CRect)

Die Klasse CTextTable

Die eigentliche Texttabelle ist nun eine Liste von Textitems:

```
class CTextTable {
    // Methoden
        CTextTable();
        ~CTextTable();

        void AddTextItem(CTextItem *pTextItem);
        void RemoveTextItem(CTextItem *pTextItem);
        CTextItem* InsertText(CString strText, DWORD dwLanguageID,
                    CTextItem *pSuggestedTextItem);
        void UpdateTextItem(CString strText, DWORD dwLanguageID,
                    CTextItem *pTextItem);

    // Attributes
        CPtrList textItems;
};
```

`AddTextItem()` fügt ein schon bestehendes TextItem der Tabelle hinzu, bzw. hängt seine Referenz in die Liste.

`RemoveTextItem()` löscht ein TextItem aus der Tabelle. Das heisst es wird aus der Liste gelöscht und zerstört. Sicherheitshalber sollte geprüft werden, ob das TextItem auch wirklich in der Tabelle vorkommt. Kommt es nicht vor wird nichts getan. Zusätzlich muss sichergestellt sein, dass das TextItem von keiner Textreferenz mehr verwendet wird, das heisst seine `References` - Liste muss leer sein. Nur in diesem Fall darf die Funktion aufgerufen werden!

Die Methode **`InsertText()`** fügt einen Text in die Tabelle ein. Sie wird **beim Parsen zum Aufbau der Texttabelle** verwendet. Zwei Fälle müssen berücksichtigt werden:

- Ist der Pointer `pSuggestedTextItem` NULL, so existiert noch kein entsprechendes TextItem für diese Stelle. Das bedeutet, dass der Parser an eine Stelle im Baum gelangt, die er bis jetzt noch nicht besucht hat (für Details siehe Parsing).
- pSuggestedTextItem ist nicht NULL. Das heisst, der Parser hat diesen Knoten schon mal besucht und diese Textreferenz an das TextItem *pSuggestedTextItem gebunden.

Hier nun der Pseudocode für den Algorithmus:

```
if (pSuggestedTextItem == NULL) {
    for every TextItem t in Texttable do {
        if (t hat genau ein Eintrag in der LanguageMap und
            dieser entspricht dem Paar (dwlanguageID, strText))
                return &t;
    }
    Kreiere neues TextItem n und schreibe das Paar
    (dwlanguageID, strText) in seine LanguageMap
    füge n der Liste textItems hinzu
    return &n;
}
else {
    CString *pstrText;
    pstrText = pSuggestedTextItem->GetText(dwLanguageID);
    if (pstrText == NULL) {
        pSuggestedTextItem->SetText(strText, dwLanguageID);
        return pSuggestedTextItem;
    }
    if (*pstrText == strText) {
        return pSuggestedTextItem;
    // else
    for every TextItem t in Texttable do {
        if (pSuggestedTextItem.MatchLanguages(t))
            return &t;
    }
    Kreiere ein neues TextItem n wie folgt:
    CTextItem n = CTextItem(*pSuggestedTextItem);
    // LanguageMap wird kopiert!
    n.SetText(strText, dwLanguageMap);
    füge n der Liste textItems hinzu
    return &n;
}
```

Die Methode **`UpdateTextItem()`** dient zum Verändern eines Spracheintrages in einem TextItem **nach dem Aufbau der Tabelle**. Die Funktion kann auf unterschiedliche Weise reagieren, je nach dem wieviele der Betroffenen Textreferenzen markiert sind. Dabei müssen zwei Fälle unterschieden werden:

- alle in der `References` - Liste des TextItems `*pTextItem` vermerkten Textreferenzen sind markiert resp. durch das TreeCtrl - Fenster selektiert. In diesem Fall kann das Wort abgeändert werden, da die Änderung alle Textreferenzen betrifft. Es entsteht ein neues TextItem. Darauf folgend muss die TextTabelle durchsucht werden, ob schon ein gleiches TextItem existiert. Wenn ja, werden beide verschmolzen. Andernfalls wird das neue TextItem in die Texttabelle eingüfugt.
- nicht alle Textreferenzen der Liste sind markiert. Nur der Text für die selektierten Textreferenzen muss verändert werden. Eine Kopie des alten TextItems wird erstellt. Die nicht selektierten Textreferenzen benutzen nach wie vor das alte, die selektierteb Textreferenzen aber sollen nun das neue TextItem verwenden.

Hier nun wieder der **Pseudocode** für diese Funktion:

```
if (every textreference t ∈ pTextItem->References is selected) {
    pTextItem->SetText(strText, dwLanguageID);
    for all TextItems t in the List textItems do {
        if (pTextItem->MatchLanguages(t)) {
            for all textreferences pt in pTextItem->References do {
                pt->SetReference(pTextItem);
                    // die betroffenen Referenzen umbiegen
                füge pt der Liste t->References hinzu
                    // alle Referenzen gehören zum neuen TextItem
            }
            RemoveTextItem(pTextItem);
            //lösche *pTextItem, denn wird nicht mehr gebraucht
            return; // fertig!
        }
    }
    // wenn kein gleiches gefunden wird ist man fertig
else {
    Kreiere eine Kopie n von *pTextItem  //(wie in InsertText()
    n->SetText(strText, dwLanguageID);
    for all pTextRef in the List pTextItem->References do {
        if (pTextRef->IsSelected()) {
            lösche pTextRef in pTextItem->References
            n.AddReference(pTextRef);
            pTextRef->SetReference(&n);
                // Referenz der Textreferenzen werden umgebogen
        }
    }
    // das alte TextItem *pTextItem ist schon in Ordnung
    // das neue (n) muss noch in die Tabelle eingefügt werden
    // (gleiche Prozedur wie oben)
    for all TextItems t in the List textItems do {
        if (n.MatchLanguages(t)) {
            // gibt es ein gleiches TextItem, so werden sie vereint
            for all text references pt in n.References do {
                pt->SetReference(&t);
                    // die betroffenen Textreferenzen umbiegen
                füge pt der Liste t->References hinzu
                    // alle Referenzen gehören zum neuen TextItem
            }
            RemoveTextItem(&n);
            //lösche n, denn wird nicht mehr gebraucht
            return; // fertig!
        }
    }
    // kein gleiches gefunden
    AddTextItem(&n);
        // das neue TextItem wird in die Tabelle eingefügt
}
```

Der Resourcen Baum

Der Resourcen Baum speichert die Informationen und die Struktur der Resourcen. Wichtig zum Verständnis sind folgende zwei Punkte:

- Ein Resource - Element (Dialog, Menu, Bitmap, etc) ist im Baum immer nur einmal vorhanden.
- Alle Texte werden in der Texttabelle gespeichert.

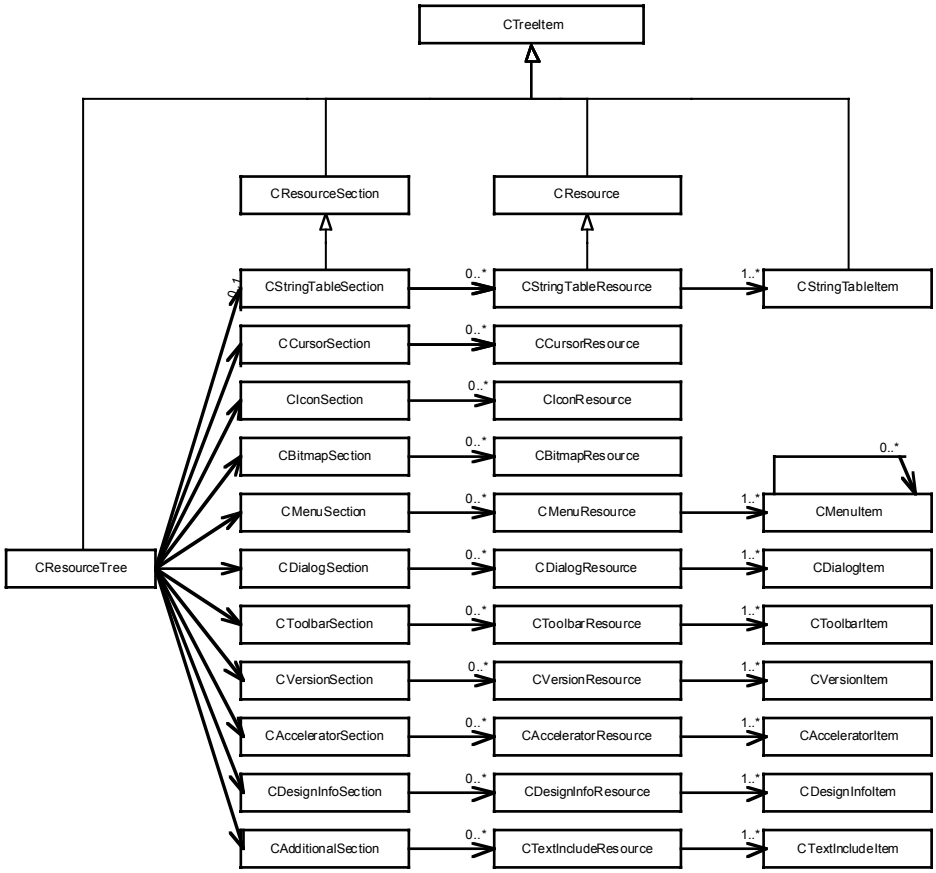
Zum ersten Punkt ist folgendes zu sagen. Betrachtet man zB. eine zweisprachige Resource - Datei, so kommen alle Resource - Elemente, die zweisprachig sind auch zweimal vor. Einmal für die eine Sprache, das zweite Mal für die zweite Sprache. DIe Struktur beider Vorkommnisse ist aber absolut identisch. Deshalb ist es nicht notwendig, die Struktur zweimal zu speichern. Im Resourcen Baum ist das so gelöst, das jede Struktur die (mind.) einmal vorkommt, in den Baum aufgenommen wird. In der Struktur selbst wird dann nur vermerkt, für welche Sprachen sie existent ist.

Zum zweiten Punkt: Die eigentlichen Texte werden in der Texttabelle gespeichert. Der Baum enthält nur Referenzen auf die Texte. Konkret sieht das so aus. An der Stelle des Baumes wo eigentlich ein Text stehen sollte, ist lediglich eine Referenz auf das entspr. TextItem. Die TextItems wiederum

merken sich alle Elemente des Baumes, die dieses TextItem referenzieren. Dies muss bei allfälligen Operationen auf dem Baum oder der Texttabelle berücksichtigt werden !

Bevor wir nun aber die einzelnen Klassen des Resourcen Baumes genau anschauen, betrachten wir zuerst eine Zusammenstellung in Form eines Klassendiagramms.

Klassendiagramm



Erläuterungen zum Klassendiagramm

Der Baum wird gebildet durch die in der Zeichnung grau schattierten Klassen gebildet. Die Wurzel des Baums bildet **`CResourceTree`**. Ein Objekt der Klasse `CResourceTree` hat maximal eine Instanz der anderen Klassen mit Namen `CXXXSection`, also `CIconSection`, `CBitmapSection`, ... , `CDesignInfoSection`. (Dies gilt für den Moment. Es wäre durchaus denkbar, dass in einer späteren Version mehrere Sections eines Types erlaubt wären; und das sogar ihre Reihenfole beliebig sein dürfte. Doch im Moment soll nur gerade die Syntax akzeptiert werden, die im Flussdiagramm in Anhang A definiert ist). Diese Sections (`CXXXSection`) können je nach Typ noch weitere Unterklassen bzw. Unterbäume haben. Dies gilt zum Bsp. für `CMenuSection`. Das Objekt dieser Klasse enthält alle in den Ressourcen vorkommenden Menus. Jedes einzelne Menu ist in einem Objekt der Klasse `CMenuResource` enthalten. Dieses wiederum enthält die eigentlichen Menuitems, welche wiederum `CMenuItems` enthalten können (Popupmenus).

Jedes `CTreeItem` - Objekt, und somit jedes Baumelement, hat eine Liste, welche die Referenzen der Sohnobjekte speichert. Diese Liste kann z.B. von `CPtrList` abgeleitet werden oder aber man verwendet direkt `CPtrList`.

Der Resourcen Baum ist natürlich inhomogen. Das heisst er besteht aus Objekten von verschiedenen Klassen. Damit er aber in gewisser Weise homogen

behandelt werden kann, ist **jedes seiner Elemente** von der Klasse **`CTreeItem`** abgeleitet (direkt oder indirekt). Diese Klasse stellt einige abstrakte Methoden zur Verfügung (Sohn-Vater-Informationen, Baumoperationen). Zur Vereinfachung sind nicht alle Vererbungsbeziehungen eingezeichnet, doch sollte dies aus der Nomenklatur ersichtlich sein (alle Sections sind von CResourceSection abgeleitet etc.)

Die Aufteilung in `CResourceSection` und `CResource` wurde gewählt, um gewisse Einheitlichkeiten zu vereinfachen. Beispielsweise kann für alle jede Resourcen eine Loadmem-Liste angegeben werden.

Die Klasse CTreeItem

Einige Methoden der Klasse sind reinvirtuell bzw. abstrakt. Hier die Definition:

```
class CTreeItem {
       CTreeItem();
       ~CTreeItem();

    // Methoden

        virtual    BOOL IsSelected();
        virtual    BOOL SetSelection(BOOL bValue);
        virtual    BOOL IsLanguageSet(DWORD dwLanguageID);
        virtual    BOOL SetLanguage(DWORD dwLanguageID);
        virtual    BOOL RemoveLanguage(DWORD dwLanguageID);
        virtual    int  NumberOfLanguages(DWORD dwLanguageID);
        virtual    BOOL HasAllTexts(DWORD dwLanguageID);
        virtual    BOOL IsComplete(DWORD dwOrigLangID,
                                   DWORD dwTransLangID);
       (virtual)   void Complete(DWORD dwLanguageID);
        virtual    BOOL ReduceTree(DWORD dwOrigLangID,
                                   DWORD dwTransLangID);
        virtual    BOOL Delete(DWORD dwLanguageID);

        virtual    void UpdateAllTexts(CTextTable *lpTextTable);

        virtual    CTreeItem*
                      StructureAlreadyExists(CTreeItem* lpOther);
        virtual    void Merge(CTreeItem * lpOther);
        virtual    void StrongCompare(CTreeItem * lpOther);
        virtual    void Assign(CTreeItem * lpOther);

    // Attributes
        CLanguageList LanguageList;
        CTreeItemList sons; // list fo the sons of this object
        BOOL bIsSelected;

       CTextItem* m_lpTextItem;
};
```

Sprachliste

Wie schon gesagt, wird für jede Resource (z.B. Dialog) nur ein Objekt im Baum erzeugt, egal in wievielen Sprachen die Resource existiert. Welche Sprachen existieren, ist nicht in CTreeItem definiert, sondern in den referenzierten CTextItems. Einige CTreeItems haben keine (DesignInfo-Resourcen), die meisten eine (z.B. CStringTableItem) und einige sogar mehrere solche Referenzen, je nachdem wie viele Aspekte eines Resourcen-Items sprachabhängig sein können. `CTextItem` wurde bereits weiter oben behandelt. eine Liste, die diejenigen Sprachen angibt, für welche der Dialog existiert.

Methoden

Die meisten der Methoden der Klasse `CTreeItem` haben rekursiven Charakter, sie arbeiten auf dem ganzen Teilbaum dessen Wurzel das aufrufende Objekt ist. Deshalb sind diese Methoden virtuell und müssen für alle Baumklassen überschrieben werden.

Die Bedeutung der Methode `StructureAlreadyExists()` wird weiter unten diskutiert, ebenso die Methoden `Merge(), Assign()` und `StrongCompare()`. Sie sind für die korrekte Behandlung einer Resource, die in mehreren Sprachen vorkommt, verantwortlich.

`IsSelected()` returniert den Wert der Membervariable `bIsSelected`, **`SetSelection()`** setzt ihren Wert.

Die Methode **`IsLanguageSet()`** prüft ob die Sprache mit ID `dwLanguageID` in der Sprachliste eingetragen ist. **`SetLanguage()`** fügt eine Sprache der Liste hinzu und **`RemoveLanguage()`** löscht eine Sprache aus der Liste. **`NumberOfLanguages()`** retourniert die Anzahl Sprachen in der Liste.

Die eigentlichen Texte befinden sich innerhalb der Texttabelle. Während dem Parsen werden diese Texte trotzdem (temporär) in den jeweiligen CTreeItems gespeichert (s. Parsen) . Die Methode **`UpdateAllTexts()`** überträgt die entspr. Texte oder andere sprachabh. Attribute in die Texttabelle.

Die Methode **`HasAllText()`** gibt an ob für eine bestimmte Sprache schon alle Texte erfasst worden sind.

`IsComplete()` vergleicht die Struktur des Baumes für zwei Sprachen. Die Methode gibt genau dann `TRUE` zurück, wenn für das Baumelement und alle seine Unterelemente gilt : Ist in der Sprachliste des Elements die Sprache `dwOrigLangID` enthalten, so ist auch `dwTransLangID` enthalten. In anderen Worten ist ein Teilbaum genau dann komplett, wenn jedes seiner Elemente, das in der Originalsprache existiert auch für die Übersetzungssprache existiert. **`Complete()`** komplettiert ein Teilbaum für die Übersetzungssprache gemäss der Originalsprache. D.h. für alle Elemente des Teilbaumes, deren Sprachliste die Sprache `dwOrigLangID` enthält, wird auch die Sprache `dwTransLangID` aufgenommen (sofern sie noch nicht vorhanden ist in der Sprache). Der Algorithmus sieht ungefähr so aus:

```
If (IsLanguageSet(dwOrigLangID)) SetLanguage(dwTransLangID);
for all TreeItems *pt in the List sons do {
    pt->Complete();
}
```

`ReduceTree()` reduziert den Baum für die Übersetzungssprache gemäss der Originalsprache. D.h. für alle Elemente des Teilbaumes, deren Sprachliste die Sprache `dwOrigLangID` nicht enthält, wird auch die Sprache `dwTransLangID` entfernt. (sofern sie überhaupt vorhanden war).

`Delete()` löscht einen Teilbaum für eine bestimmte Sprache mit ID `dwLanguageID`. Das bedeutet, dass in jedem Element des Teilbaumes wird die Sprache aus der Sprachliste entfernt (sofern überhaupt vorhanden).

Alle diese Methoden werden weiter unten im Einzelnen beschrieben. Dabei wird nicht die genaue Implementation im Vordergrund stehen. Hauptpunkt ist das Zusammenspiel dieser Methoden bei Abläufen wie Parsen, Niederschreiben in Dateien und so weiter.

Bis hier sind lediglich die Klassen für die Laufzeitrepräsentation einer Resource beschrieben. Im folgenden werden Klassen und Funktionalitäten beschrieben, die für das Parsen der Resourcen-Dateien und den Aufbau des Resourcenbaums und der Texttabelle zuständig sind.

Parsing

Grundsätzliches

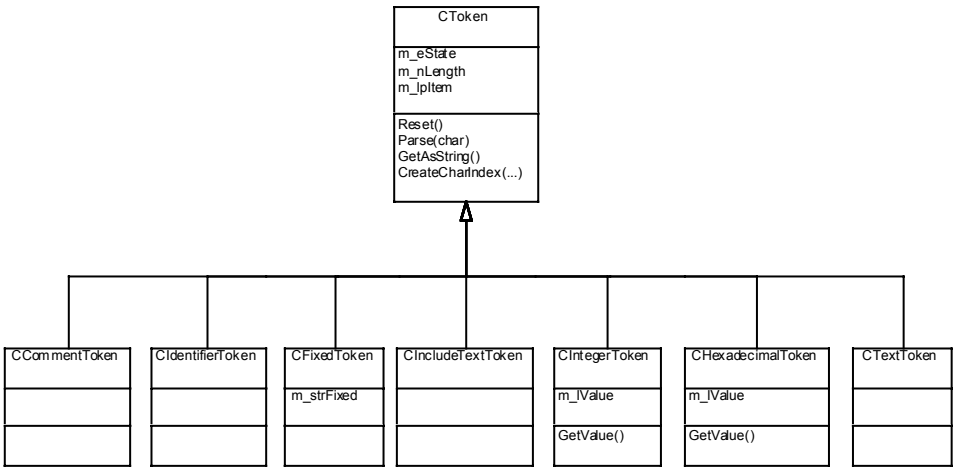
Das Parsen der Resourcendateien ist (soweit wie möglich) separiert, d.h. das Parsen wird nicht in die CTreeItem und deren Derivate integriert, sondern geschieht über eine zusätzliche (temporäre) Instanz von CParser (s. unten).

Lexikalische Analyse

Für die lexikalische Analyse stehen die Erben der Klasse CToken zur Verfügung (s. Grafik). Ein Token (eine Instanz) soll lediglich eine bestimmte Zeichenfolge erkennen und ggf. zusätzliche Informationen speichern.

Die Resourcen-Dateien weisen eine C-ähnliche Syntax auf, weshalb die Aufteilung folgendermassen gewählt wurde:

Token	Erkennung	Beispiele
CIdentifierToken	Bezeichner	IDC_BUTTON1, IDS_BLABLA
CIntegerToken	Dezimale Ganzzahlen	100, 203, 0L, 0UL
CHexadecimalToken	Hex-Zahlen	0xabcdef, 0x00000001L
CFixedToken	reservierte Wörter	DIALOG, #define, *
CTextToken	Strings	„&Open File“
CIncludeTextToken	Strings in eckigen Klammern	<afxres.h>
CCommentToken	C-Kommentare	// einzeilger Kommentar /* Kommentar */



Klassendiagramm der Tokens

Die abstrakte Klasse CToken stellt virtuelle Methoden für das Parsen eines einzelnen Zeichens sowie Statusabfragen zur Verfügung. Ein Token ist in diesem Sinne eine Zustandsmaschine, die mit Zeichen gefüttert wird. Hier die wesentlichen Methoden von CToken:

```
class CToken
{
  // Types
  public:
    // ParseZustände :
    typedef enum { statePossible,
                   stateTerminated,
                   stateBad }
                   ParseState;

  // Constructors / Destructors
  public:
                              CToken(BOOL bMayBeLonger);
    virtual                    ~CToken();

  // Methods
  public:
           CToken::ParseState  GetState(void) const
           CToken::ParseState  GetLastState(void) const;
           int                 GetLength(void) const;
    virtual void               Reset(void);

            BOOL               MayBeLonger(void)const;

    virtual CToken::ParseState Parse(const int);
    virtual CString            GetAsString(void) const = 0;
    virtual CString            GetTokenInfo(void)const = 0;

  // Attributes
  protected:
    int                 m_nLength;
    CToken::ParseState  m_eState,
                        m_eLastState;
    BOOL                m_bMayBeLonger;
};
```

Ein Token kennt drei Zustände:

Zustand	Bedeutung
statePossible	Token nach wie vor möglich, aber noch nicht terminiert (ist ebenfalls Initialzustand).
stateTerminated	Token fertig erkannt (unter gewissen Umständen kann das Token aber noch „weiterwachsen“, z.B. bei Ganzzahlen)
stateBad	Token nicht mehr möglich

Die Methoden sind i.d.R. selbsterklärend. Wesentlich und sind die Methoden **`GetState()`** und **`GetAsString()`** (bzw. **`GetValue()`** in einigen Erben). Jene liefert den aktuellen Zustand, diese das erkannte Token als String.

Syntaktische Analyse

Für die syntaktische Analyse existiert die Klasse **`CParser`**. Sie definiert noch keine Syntax, stellt aber die nötigen Methoden zur Verfügung:

```
class CParser
{
  // Constructors / Destructors
  public:
                      CParser();
    virtual           ~CParser();

  // Methods
  public:
    // Verwaltung / Informationen

            void      SetStream(istream& is);

            int       GetLineNumber(void)const;
            int       GetColumn(void)const;
            long      GetStreamPos(void)const;
            BOOL      Finished(void)const;

    virtual void      Parse(void);
    virtual void      Init(void);

  protected:
    virtual void      RegisterTokens();
            void      AddToken(CToken *lpToken);

    // SYNTAKTISCHE ANALYSE
    BOOL    CheckOrFail(const BOOL bCheck, const CString& strError);
    BOOL    CheckToken(const CToken *lpToken );
    BOOL    CheckTokenOrFail(const CToken *lpToken);
    BOOL    CheckAndAdvance(const CToken *lpToken);
    CToken *GetNextToken();
    CToken *GetRecognizedToken();

    virtual void InvokeParseError(const CToken *lpToken);
    virtual void InvokeParseError(const CString &strError);

  private:

   CPtrList m_apTokenList,       // alle Tokens
            m_apActualTokenList, // possible und terminated-Tokens
            m_apLastTokenList;   // dito für letzten Durchgang


   CParseStrea         m_ParseStream;

   int                 m_bReparse;

   CMapCharToTokenList m_MapCharToTokenList;

};
```

Alle Tokens befinden sich in der Liste **`m_apTokenList`**. Zu Beginn eines neuen Parsdurchgangs werden diese zurückgesetzt (**`statePossible`**); die Liste wird nach **`m_apActualTokenList`** kopiert.

Alle in **`m_apActualTokenList`** befindlichen Tokens werden mit einem (von irgendeinem Stream) eingehenden Zeichen gefüttert. Diese wechseln ihre Zustände. Die Liste wird nach **`m_apLastTokens`** (s. unten) kopiert und von stateBad-Tokens bereinigt. Nun kann der Terminierungstest erfolgen. Ein Token ist erkannt, wenn es als einziges terminiert ist (einziges Token in der Liste mit stateTerminated) und kein anderes möglich. Wie wir oben gesehen haben, ist ein Token in diesem Zustand aber nicht unbedingt fertig. Hierzu wird statt eines look-aheads ein „look-back“ benutzt: Sind im aktuellen Durchgang alle Tokens stateBad, dann wird überprüft, ob im letzten Durchgang ein Token terminiert war. Die List m_apLastTokens verweist auf alle Tokens, die im **letzten** Durchgang statePossible oder stateTerminated waren. In diesem Fall ist für den nächsten Durchgang ein erneutes Parsen des eben „geschluckten“ Zeichens nötig (**`m_bReparse = TRUE`**).

Ist noch kein Token terminiert, erfolgt ein neuer Durchgang.

Der Parser benutzt als Eingabe-Stream eine Klasse **CParseStream**. Diese wird auf einen beliebigen Stream aufgesetzt und dient lediglich zum zählen der Zeilen und Spalten - um sinnvolle Fehlermeldungen zu generieren.

Nachtrag:

Da die Geschwindigkeit bei der ersten Implementation etwas zu wünschen übrig liess, wurden folgende Anpassungen vorgenommen:

- Der Parser verfügt nun über einen Index namens **CMapCharToTokenList**. Dieser Index ist, wie der Name schon sagt, eine Multi-Map von Characters nach Token(s).
- Jedes Token hat eine Methode **CreateCharIndex()**, die das Token in der Multli-Map für alle Zeichen einträgt, womit das Token beginnen kann (z.B. 0..9 für Integer)
- Der **CParseStream** verfügt über eine Methode **EatWhite()**, die den Stream bis ans nächste Nicht-WhiteSpace-Zeichen positioniert.
- Bei der Instanzierung des Parsers wird der Index erstellt. Bei jedem neuen Durchgang wird ein **EatWhite()** ausgelöst. Mit dem nachfolgende Zeichen und dem Index werden die in Frage kommenden Tokens bestimmt. Vorteil: Es müssen beim ersten Durchgang nicht alle Tokens getestet werden.

Aufbau einer Syntax und Semantik

Um eine Syntax zu definieren erbt man von dieser Klasse, registriert die zu erkennenden Tokens im Konstruktor und definiert weitere Methoden, die die Syntax nachbilden. Der Einsprungpunkt bildet die Methode **Parse()**.

Mittels **GetNextToken()** wird das nächste Token geholt, mittels den **Check**-Methoden kann verglichen und ggf. eine Fehlermeldung ausgelöst werden.

Hierzu ein Beispiel:

Annahme: die zu parsende Datei habe die Struktur

```
BEGIN
  IDENTIFIER = INTEGER
  IDENTIFIER = INTEGER
  ...
END
```

Wir definieren dazu drei **CFixedToken** für "BEGIN", "END" und "=",ein **CIntegerToken** und ein **CIdentifierToken** (als Member einer neuen Klasse **CMyParser**),

```
CMyParser::CMyParser() :
  m_Begin(„BEGIN"),
  m_End(„END"),
  m_Equals(„="),
  m_Ident(),
  m_Int()
{
}
```

und registrieren diese in der **m_apTokenList** (innerhalb der überschriebenen **RegisterToken()**-Methode mittels **AddToken()**):

```
void CMyParser::RegisterTokens()
{
  AddToken(&m_Begin);
  AddToken(&m_End);
  AddToken(&m_Equals);
  AddToken(&m_Ident);
  AddToken(&m_Int);

  CParser::RegisterTokens();
}
```

Nun benötigen wir für die IDENT-EQUALS-INT-Sequenz lediglich eine enpsr. Methode, die die Tokens überprüft und natürlich die Informationen speichert, z.B:

```
BOOL CMyParser::Assignment()
{
   if(CheckToken(&m_Ident)
   {
      CString aIdent =
          m_Ident.GetAsString()     // Info speichern
      GetNextToken();               // nächstes Token
      CheckTokenOrFail(&m_Equals);  // dieses MUSS ein "=" sein
      GetNextToken();               // nächstes Token
      CheckTokenOrFail(&m_Int);     // MUSS ein Integer sein !
      long aInt = m_Int.GetValue()  // Info speichern
      GetNextToken();               // nächstes Token (nötig
                                    // für nächsten Durchgang)

      //
      // mach' irgendwas mit aIdent und aInt ...
      //

      return TRUE                   // Wir hatten eine Zuweisung
   }
   return FALSE                     // keine Zuweisung
}
```

Der Einsprung für Parse sieht dann so aus:

```
void CMyParser::Parse();
{
   GetNextToken();
   CheckTokenOrFail(&m_Begin);
   GetNextToken();
   while(Assignment());             // die Sequenz !
   CheckTokenOrFail(&m_End);
                                    // fertig!
}
```

Anschliessend evtl. noch Aufräumarbeiten.

Hinweis: In diesem Beispiel wird alles nach "END" ignoriert!

Aus diesem Beispiel sollte klar werden wie ein Parsen in etwa funktioniert. Die Semantik wird ebenfalls in solche Syntax-Methoden eingebettet

Parsen der Resourcendateien

Die Struktur der Resourcen-Dateien kann den Syntaxdiagrammen entnommen werden. Im wesentlichen sind die Dateien nach Sprachen aufgetrennt (Language-Sections), die dann die einzelnen Resourcen enthalten.

Das Parsen erledigt die Klasse **`CResourceParser`**. Sie erkennt und instanziert Resourcen vollständig, d.h. inkl. den gefundenen Texten. Diese

Instanzen werden zur weiteren Verwaltung an eine CResourceTree-Instanz weiterdelegiert (Verschmelzen von Resourcen, Einträge in die Texttabelle).

Hier ein Beispiel für eine einfache Resource (Stringtable):

```
void CResourceParser::StringTable()
{
   if(CheckTokenOrFail(&STRINGTABLE))
   {
      GetNextToken();
      LoadMem();
      while(OptionalStatements());
      m_strResourceID.Empty();
      m_lpStringTableResource = new CStringTableResource;
      m_lpActualResource = m_lpStringTableResource;
      SetResourceSettings(m_lpStringTableResource);
      CheckTokenOrFail(&BEGIN);
      GetNextToken();
      while(StringTableEntry());
      CheckTokenOrFail(&END);
      GetNextToken();
      m_lpResourceTree ->
            AddStringTableResource(m_lpStringTableResource);
      m_lpStringTableResource = NULL;
   }
}
```

Erläuterungen:

- **`LoadMem()`**, **`OptionalStatements()`**, sowie **`StringTableEntry()`** sind Methoden für die syntaktische Analyse, die Attribute von **`m_lpStringTableResource`** setzen bzw. vorbereiten.

- **`m_lpStringTableResource`** ist die neu entstandene Resource, die nach dem Aufbau mittels **`AddStringTableResource()`** an **`m_lpResourceTree`** geschickt wird. Das anschliessende = NULL verdeutlicht die Delegation der Verantwortlichkeit.

Für alle möglichen Resourcen und Resourcenattibute gibt es im CResourceParser Felder, wie z.B. Abmessungen eines DialogControls (**`m_Rectangle`**) oder die Loadmem-Liste einer Resource (**`m_aLoadMemList`**). Die Zwischenspeicherung dieser Informationen ist deshalb notwendig, da die Tokens nach einem Durchgang ihre Informationen verlieren.

Einmaligkeit der Strukturen

Es wurde bereits gesagt, dass zB. ein Menu nur dann in einem neuen Objekt gespeichert wird, wenn es nicht schon Objekt gibt, dass das gleiche Menu (MenuID!) beinhaltet. Andernfalls wird einfach das bestehende Menu ergänzt.

Hier kommt nun die Arbeit des Resourcenbaums zum Zuge.

Die Klasse CResourceTree besitzt für jeden Resourcen-Typ eine entsprechende AddResource()-Methode, welche nichts anderes tut als die erhaltene Resource in die ensprechende CResourceSection einzutragen. Dies geschieht mittels der AddSon()-Methode, die für CResourceSection - im Hinblick auf die Einmaligkeit der Strukturen - folgendermassen angepasst wurde (Pseudo-Code).

```
CResourceSection::AddSon(CTreeItem* lpSon)
{
   if(StructureAlreadyExists(lpSon))
   {
        lpExistingStructure = ...  // Struktur ausfindig machen

        // Strukturen vergleichen (ohne Text)

        if((!lpExistingStructure->StrongCompare(lpSon)) &&
            (ShallIHomogenize()))
        {
            lpExistingStructure->Assign(lpSon);
        }
        // falls nicht gleich : Struktur fürs homogenisieren
        //                      auswählen und entspr. zuweisen

        //Strukturen Verschmelzen
        lpExistingStructure->Merge(lpSon)

        // anschliessend : neue Texte eintragen (neue Sprache für
        //                 diese Resource)
        UpdateAllTexts(m_lpTextTable);

        // Destruktion des übergebenen Objekts !
        delete lpSon;
   }
   else
   {
       // Texte eintragen (in TextTable)

       CTreeItem::AddSon(lpSon);
       // normales AddSon (d.h. Resource wird in diese Section
       //                  aufgenommen):
   }
}
```

Wie bereits erwähnt enthalten die Resourcen nach dem Parsen noch alle Texte. Der Eintrag in die Texttabelle erfolgt also hier!

Strukturen verschmelzen bedeutet hier nichts anderes als die Zuweisung der Texte des übergebenen Objekts, d.h. ein Update der temporären Texte für einen anschliessenden in die Texttabelle. Diese Methode kann rekursiven Charakter haben, z.B. für Menus: ein Verschmelzen von Menu löst ein Verschmelzen von MenuItems auf tieferen Ebenen aus, falls ein **`StructureAlreadyExists()`** für diese Items positiv beantwortet werden kann. Die Methode **`CTreeItem::Merge()`** hat deshalb ähnliches Aussehen wie obige **`AddSon()`**-Methode.

Das Problem der Leerzeichen, Tabulatoren, Zeilenumbrüche und Kommentaren

Leerzeichen, Tabulatoren, Zeilenumbrüche und Kommentare aus zwei Gründen gewisse Probleme mit sich:

1. Ein Anforderung an die Applikation ist die folgende: wird eine Resourcedatei gelesen und dann sogleich, ohne zu verändern, in einer neuen Datei gespeichert, so müssen Original und "Kopie" so weit als möglich **identisch** sein. Das bedeutet das diese Zeichenfolgen nicht nur einfach übergangen werden können, denn sie müssten gespeichert werden.
2. Da Kommentare keine Kontextinformationen beinhalten (auf was bezieht sich der Kommentar?), wird durch die obige Forderung die Grammatik wesentlich komplexer.

Um das Problem etwas zu vereinfachen wurde folgendermassen vorgegangen:

- MS Developer Studio generiert die Resourcen nach einem bestimmten Muster, so dass die Struktur (insbesondere die Kommentare) immer gleich

aussieht. Statt diese Kommentare zu speichern und zu versuchen, sie den richtigen Sections bzw. Resourcen zuzuordnen (leider nicht gelungen!), sind die Kommentare nun fixer Bestandteil der Serialisierung. Weiter existiert eine Pseudo-Resource names TEXTINCLUDE, die für die Speicherung zusätzlicher Informationen zuständig ist (Defines, Includes u.ä.). Es genügt deshalb, diese Informationen korrekt in den Serialisierungsprozesses einzubinden.

- Statt die ganze Formatierung des Resourcen-Skripts mitzuspeichern, wird bei der Serialisierung eine Hilfsklasse benutzt, die Einrückungen und NewLines automatisch vornimmt (s. **`CAutoIndentStream`**).

Hilfsklassen

CLanguageTable

Sie speichert Informationen über alle verfügbaren Sprachen (ID-Werte, Kommentare, Defines für die Language-Sections, etc). Von ihr gibt es nur eine Instanz.

CAutoIndentStream

Diese Klasse ist ein Stream-Aufsatz und dient zur formatierten Ausgabe:

```
class CAutoIndentStream
{
  // Constructors / Destructors

  public:
              CAutoIndentStream(ostream& os, size_t nWidth);
             ~CAutoIndentStream();

  // Methods
  public:
     void     SetPageWidth(const size_t nWidth);
     void     SetPosition(const size_t nPosition);
     void     SetIndent(const size_t nIndent);
     void     SetNewLineEndString(const CString & strNewLineEnd);
     void     ChangeIndent(const int nChange);
}
```

Die Methoden erklären sich praktisch von selbst. Nachdem man den Output-Stream angegeben und die Formatierung gewählt hat, kann man dieses Objekt wie einen gewöhnlichen Stream ansprechen (mittels **<<**).

Datenbankmanager

Der Datenbankmanager muss zwei Dinge tun können. Erstens muss er für ein Wort in einer bestimmten Sprache alle möglichen Übersetzungen in einer anderen Sprache finden können, zweitens soll er vom Benutzer erstellte Übersetzungen in die Datenbank eintragen.

Dies wird mit folgenden zwei Klassen gelöst:

CDictionary

Diese Klasse dient zum Abfragen von Übersetzungen. Der Dictionary wird mittels Generate(wOrigLanguage, wDestLanguage) komplett aufgebaut, d.h.

Datenbankzugriffe erfolgen nur hier. Die Methode Lookup(strText) liefert eine String-Liste mit allen möglichen Übersetzungen zurück. Um die Performance.

CDictionaryUpdater

Im Konstruktor werden wiederum Original und Übersetzungssprache festgelegt. Mittels der Methode **`AddNew(strOrig, strDest)`** wird eine neue Übersetzung eingetragen.

Datenbank

Die Datenbank soll in MS Access 7.0, das heisst im .mdb - Format realisiert werden. Die Datenbankdefinition muss mit derjenigen von WMLS übereinstimmen.

Der Zugriff soll über die Schnittstelle DAO (Direct Access Objects) erfolgen. Die oben beschriebenen Klassen benutzen dazu ein Derivat der Klasse CDaoRecordSet, eine MFC-Klasse.

Die DatebDie Datenank besteht lediglich aus einer Tabelle mit den folgenden Feldern:

Feld Name	Typ
LeftText	String
LeftLangID	Integer
RightLangID	Integer
RightText	String

Dabei soll LeftLangID immer kleiner als RightLangID sein. Die Tabelle funktioniert also in beide Richtungen und für ein bestimmtes Sprach-Paar ist die Ordnung damit festgelegt.

Anhang A: Syntaxdiagramme

Hier ist das Syntaxdiagramm einzufügen!

WMLS ResourceWizard

Syntaxdiagramme

By TeTrade AG

Version 1.0

This manual was produced using *Doc-To-Help*®, by WexTech Systems, Inc.

WexTech Systems, Inc.
310 Madison Avenue, Suite 905
New York, NY 10017
+1 (212) 949-9595
Fax: +1 (212) 949-4007

Inhalt

Syntaxdiagramme

Ressource Files

RES FILE

You can specify a .RES file when linking a program. The .RES file is created by the resource compiler (RC). LINK automatically converts .RES files to COFF. The CVTRES.EXE tool must be in the same directory as LINK.EXE or in a directory specified in the PATH environment variable.

RC FILE

This is a listing of all of the Microsoft Windows resources that the program uses. It includes the icons, bitmaps, and cursors that are stored in the RES subdirectory. This file can be directly edited in Microsoft Developer Studio.

RC2 FILE

This file contains resources that are not edited by Microsoft Developer Studio. You should place all resources not editable by the resource editor in this file.

Example: Using Resource Files Not Maintainable by the Visual C++ Resource Editors

The resource file CTRLTEST\RES\CTRLTEST.RC2 is an example of a resource file not maintainable by the Visual C++ resource editors in a human-readable form. If you were to open CTRLTEST.RC2 in Visual C++, and then save it, you would lose useful human-readable information, even though the Resource Compiler would still be able to compile the .RC2 file and produce an equivalent binary .RES file. Thus, RES\CTRLTEST.RC2 has been added as a #include in CTRLTEST.RC with a Compile-Time Directive specified with the Resource File Set Includes command.

Three categories of human-readable information in CTRLTEST.RC2 that are not maintainable by the Visual C++ resource editors are:

· Custom control styles symbols_ For example, MSS_VERTICAL is a style defined for the "MicroScroll32" spin control. Although Visual C++ can interpret this symbol as it reads in the .RC2 file, Visual C++ would write it back out to the .RC2 file as a hexadecimal value.

· Standard Windows WS_ or control style symbols used in a control from a standard Windows control-derived class_ For example, two standard edit

control styles_ **ES_LEFT** and **ES_AUTOSCROLL** _ are defined for the hand-edit pen control in the IDD_PENEDIT dialog. Although Visual C++ can interpret these symbols as it reads in the .RC2 file, Visual C++ would write it back out to the .RC2 files as hexadecimal value.

· Arithmetic in the .RC file_ For example, CTRLTEST.RC2 uses expressions such as "IDC_ALC_FIRST+2" to identify controls in the **IDD_PENEDIT** dialog. The symbolic expression would be written back out to the .RC2 file as a single hexadecimal value by Visual C++.

The CTRLTEST sample illustrates the pros and cons of using an .RC2 file in the case of a dialog that has a custom control with styles defined with constants in a header file. Both dialogs **IDD_WNDCLASS_EDIT** and **IDD_SPIN_EDIT** have custom controls with symbolically defined styles; but **IDD_WNDCLASS** is specified in a .RC file editable by the Visual C++ dialog editor; whereas **IDD_SPIN_EDIT** is specified in a .RC2 file that is only manually editable. The following table summarizes the pros and cons of using the .RC2 file:

Dialog Identification:	IDD_WNDCLASS_EDIT	IDD_SPIN_EDIT
Resource script defined in:	CTRLTEST.RC	RES\CTRLTEST.RC2
WNDCLASS of custom control:	"paredit"	"MicroScroll32"
Style constants defined in:	PAREDIT.H	MUSCRL32.H
Example style constant:	PES_NUMBER	MSS_VERTICAL
Editable by Visual C++?	Yes	No
Can use #define **styles?**	No	Yes

The trade-off is that if you use the .RC2 file, you can use human-readable symbolic styles defined in the header file for the custom control, but you cannot edit the .RC2 file with the Visual C++ dialog editor. It is easier to lay out the dialog using Visual C++ than it is to manually write resource script; and writing resource script is more error prone. On the other hand, the styles are not self-documenting, when displayed in hexidecimal in the custom control property page by the Visual C++ dialog editor.

CLW FILE

This file contains information used by ClassWizard to edit existing classes or add new classes. ClassWizard also uses this file to store information needed to create and edit message maps and dialog data maps and to create prototype member functions.

RC.EXE

To create a Macintosh resource file using RC, you must use the /M option. When you set this option, the default output extension for resource files is .RSC instead of .RES. This option also defines the **-MAC** preprocessor symbol.

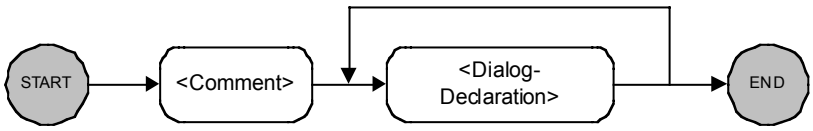
All other RC.EXE options operate identically in Visual C++ 4.0 and Visual C++ for Macintosh.

Syntax-Diagram Dialog

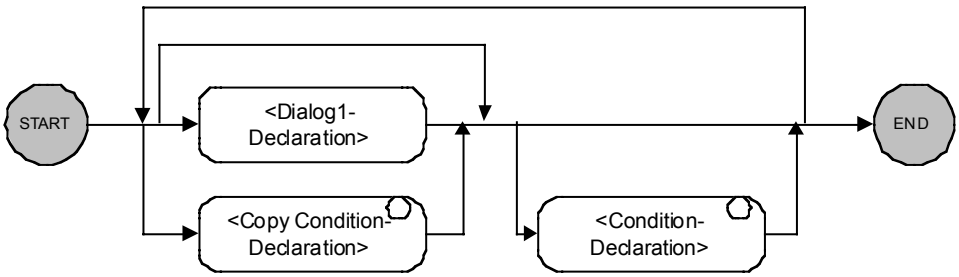
The **DIALOG** statement defines a window that an application can use to create dialog boxes. The statement defines the position and dimensions of the dialog box on the screen as well as the dialog box style.

Syntax

```
nameID DIALOG [ load-mem] x, y, width, height
[optional-statements]
BEGIN
    control-statement
    . . .
END
```

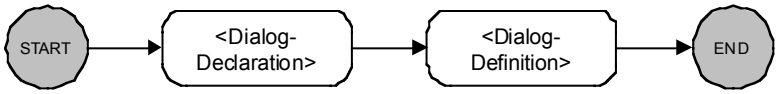


Dialog-Declaration

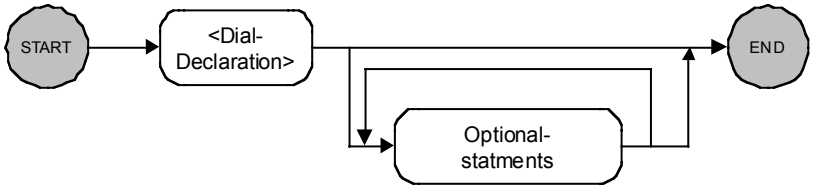


for each “Copy Condition-Declaration“ exist one “Condition-Declaration“ it Is also possible to have “Condition-Declaration“ if Dialog is made with condition

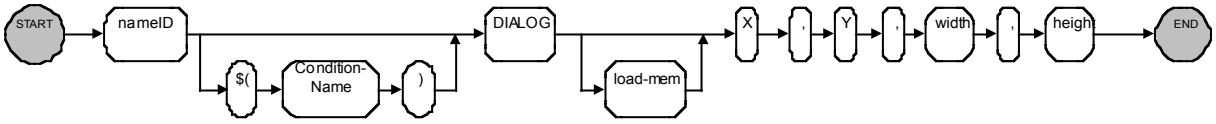
Dialog1-Declaration



Dialog-Declaration



Dial-Declaration



Parameters

nameID

Identifies the dialog box. This is either a unique name or a unique 16-bit unsigned integer value in the range 1 to 65,535.

load-mem

Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

option-statements

Specifies options for the dialog box. This can be zero or more of the following statements:

CAPTION "*text*"

Specifies the caption of the dialog box if it has a title bar. See CAPTION for more information.

CHARACTERISTICS *dword*

Specifies a user-defined double-word value for use by resource tools. This value is not used by Windows. For more information, see CHARACTERISTICS.

CLASS *class*

Specifies a 16-bit unsigned integer or a string, enclosed in double quotation marks ("), that identifies the class of the dialog box. See CLASS for more information.

LANGUAGE *language,sublanguage*

Specifies the language of the dialog box. See LANGUAGE for more information.

STYLE *styles*

Specifies the styles of the dialog box. See STYLE for more information.

The STYLE statement defines the window style of the dialog box. The window style specifies whether the box is a pop-up or a child window. The default style has the following attributes: WS_POPUP, WS_BORDER, and WS_SYSMENU.

Syntax

STYLE *style*

Parameter

style

Specifies the window style. This parameter takes an integer value or redefined name. The following lists the redefined styles:

DS_LOCALEDIT

Specifies that edit controls in the dialog box will use memory in the application's data section. By default, all edit controls in dialog boxes use memory outside the application's data section. This feature can be suppressed by adding the DS_LOCALEDIT flag to the **STYLE** command for the dialog box. If this flag is not used, EM_GETHANDLE and EM_SETHANDLE messages must not be used since the storage for the control is not in the application's data section. This feature does not affect edit controls created outside of dialog boxes.

DS_MODALFRAME

Creates a dialog box with a modal dialog box frame that can be combined with a title bar and System menu by specifying the WS_CAPTION and WS_SYSMENU styles.

DS_NOIDLEMSG

Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.

DS_SYSMODAL
Creates a system-modal dialog box.

WS_BORDER
Creates a window that has a border.

WS_CAPTION
Creates a window that has a title bar (implies the WS_BORDER style).

WS_CHILD
Creates a child window. It cannot be used with the WS_POPUP style.

WS_CHILDWINDOW
Creates a child window that has the WS_CHILD style.

WS_CLIPCHILDREN
Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.

WS_CLIPSIBLINGS
Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, this style clips all other top-level child windows out of the region of the child window to be updated. (If the WS_CLIPSIBLINGS style is not given and child windows overlap, it is possible, when drawing in the client area of a child window, to draw in the client area of a neighboring child window.) For use with the WS_CHILD style only.

WS_DISABLED
Creates a window that is initially disabled.

WS_DLGFRAME
Creates a window with a modal dialog box frame but no title.

WS_GROUP
Specifies the first control of a group of controls in which the user can move from one control to the next by using the arrow keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (that is, one group ends where the next begins). This style is valid only for controls.

WS_HSCROLL
Creates a window that has a horizontal scroll bar.

WS_ICONIC

Creates a window that is initially iconic. For use with the WS_OVERLAPPED style only.

WS_MAXIMIZE
Creates a window of maximum size.

WS_MAXIMIZEBOX
Creates a window that has a Maximize box.

WS_MINIMIZE
Creates a window of minimum size.

WS_MINIMIZEBOX
Creates a window that has a Minimize box.

WS_OVERLAPPED
Creates an overlapped window. An overlapped window has a caption and a border.

WS_OVERLAPPEDWINDOW
Creates an overlapped window having the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.

WS_POPUP
Creates a pop-up window. It cannot be used with the WS_CHILD style.

WS_POPUPWINDOW
Creates a pop-up window that has the WS_POPUP, WS_BORDER, and WS_SYSMENU styles. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the System menu visible.

WS_SIZEBOX
Creates a window that has a size box. Used only for windows with a title bar or with vertical and horizontal scroll bars.

WS_SYSMENU
Creates a window that has a System-menu box in its title bar. Used only for windows with title bars. If used with a child window, this style creates a Close box instead of a System-menu box.

WS_TABSTOP
Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style. This style is valid only for controls.

WS_THICKFRAME
Creates a window with a thick frame that can be used to size the window.

WS_VISIBLE
Creates a window that is initially visible. This applies to overlapping and pop-up windows. For overlapping windows, the y parameter is used as a parameter for the **ShowWindow** function.

WS_VSCROLL
Creates a window that has a vertical scroll bar.

Remarks

If the redefined names are used, you must include WINDOWS.H.

EXSTYLE=extended-styles

Specifies the extended styles of the dialog box. See **EXSTYLE** for more information.

VERSION dword

Specifies a user-defined doubleword value. This statement is intended for use by additional resource tools and is not used by Windows. For more information, see **VERSION**.

For more information on the *x*, *y*, *width*, and *height* parameters, see Common Statement Parameters.

Remarks

The **GetDialogBaseUnits** function returns the dialog base units in pixels. The exact meaning of the coordinates depends on the style defined by the **STYLE** option statement. For child-style dialog boxes, the coordinates are relative to the origin of the parent window, unless the dialog box has the style DS_ABSALIGN; in that case, the coordinates are relative to the origin of the display screen.

Do not use the WS_CHILD style with a modal dialog box. The **DialogBox** function always disables the parent/owner of the newly created dialog box. When a parent window is disabled, its child windows are implicitly disabled. Since the parent window of the child-style dialog box is disabled, the child-style dialog box is too.

If a dialog box has the DS_ABSALIGN style, the dialog coordinates for its upper-left corner are relative to the screen origin instead of to the upper-left corner of the parent window. You would typically use this style when you wanted the dialog box to start in a specific part of the display no matter where the parent window may be on the screen.

The name **DIALOG** can also be used as the class-name parameter to the **CreateWindow** function to create a window with dialog box attributes.

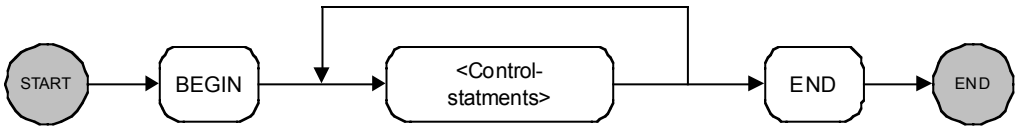
Example

The following demonstrates the usage of the **DIALOG** statement:

```
#include <windows.h>

ErrorDialog DIALOG  10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
    CTEXT "Select One:", 1, 10, 10, 280, 12
    PUSHBUTTON "&Retry", 2, 75, 30, 60, 12
    PUSHBUTTON "&Abort", 3, 75, 50, 60, 12
    PUSHBUTTON "&Ignore", 4, 75, 80, 60, 12
END
```

Dialog-Definition



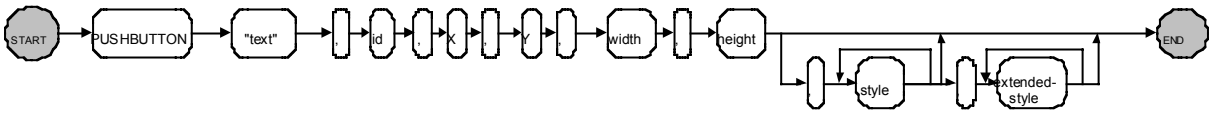
Control-statments

PUSHBUTTON

The **PUSHBUTTON** statement creates a push-button control. The control is a round-cornered rectangle containing the given text. The text is centered in the control. The control sends a message to its parent whenever the user chooses the control.

Syntax

PUSHBUTTON text, id, x, y, width, height [[, style [[, extended-style]]]]



Parameters

style
: Specifies styles for the pushbutton, which can be a combination of the BS_PUSHBUTTON style and the following styles: WS_TABSTOP, WS_DISABLED, and WS_GROUP.

The default style is BS_PUSHBUTTON and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

The following example demonstrates the use of the **PUSHBUTTON** statement:

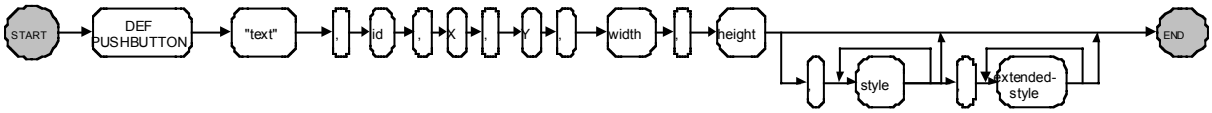
```
PUSHBUTTON "ON", 7, 10, 10, 20, 10
```

DEFPUSHBUTTON

The **DEFPUSHBUTTON** statement creates a default push-button control. The control is a small rectangle with a bold outline that represents the default response for the user. The given text is displayed inside the button. The control highlights the button in the usual way when the user clicks the mouse in it and sends a message to its parent window.

Syntax

```
DEFPUSHBUTTON text, id, x, y, width, height [,
style [, extended-style]]
```



Parameters

text
: Specifies text that is centered in the rectangular area of the control.

Style
: Specifies the control styles. This value can be a combination of the following styles: BS_DEFPUSHBUTTON, WS_TABSTOP, WS_GROUP, and WS_DISABLED.

If you do not specify a style, the default style is BS_DEFPUSHBUTTON and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

This example creates a default push-button control that is labeled “Cancel”:

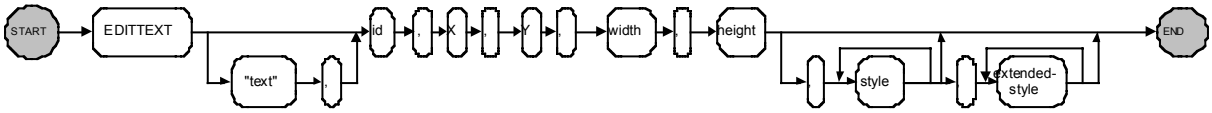
```
DEFPUSHBUTTON "Cancel", 101, 10, 10, 24, 50
```

EDITTEXT

The **EDITTEXT** statement defines an EDIT control belonging to the EDIT class. It creates a rectangular region in which the user can type and edit text. The control displays a cursor when the user clicks the mouse in it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. The user can also use the mouse to select characters to be deleted or to select the place to insert new characters.

Syntax

```
EDITTEXT text, id, x, y, width, height [[, style
[[, extended-style]]]]
```



Parameters

style
Specifies the control styles. This value can be a combination of the edit class styles and the following styles: WS_TABSTOP, WS_GROUP, WS_VSCROLL, WS_HSCROLL, and WS_DISABLED.

If you do not specify a style, the default style is ES_LEFT, WS_BORDER, and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

The following example demonstrates the use of the **EDITTEXT** statement:

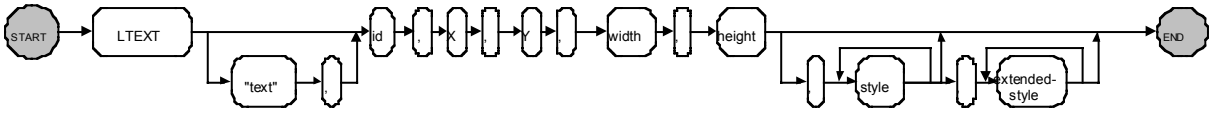
```
EDITTEXT  3, 10, 10, 100, 10
```

LTEXT

The **LTEXT** statement creates a left-aligned text control. The control is a simple rectangle displaying the given text left-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The **LTEXT** statement, which can be used only in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of the control.

Syntax

```
LTEXT text, id, x, y, width, height [[, style [[,
extended-style]]]]
```



Parameters

style

Specifies the control styles. This value can be any combination of the BS_RADIOBUTTON style and the following styles: SS_LEFT, WS_TABSTOP, and WS_GROUP.

If you do not specify a style, the default style is SS_LEFT and WS_GROUP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

This example creates a left-aligned text control that is labeled “Filename”:

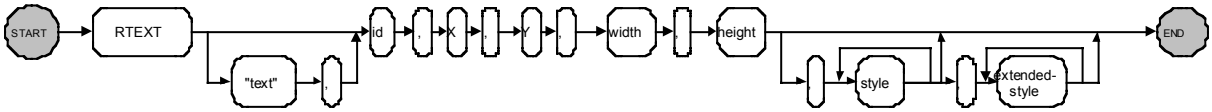
```
LTEXT "Filename", 101, 10, 10, 100, 100
```

RTEXT

The **RTEXT** statement creates a right-aligned text control. The control is a simple rectangle displaying the given text right-aligned in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

Syntax

```
RTEXT text, id, x, y, width, height [[, style [[,
extended-style]]]]
```



Parameters

style
: Specifies styles for the text control, which can be any combination of the following: WS_TABSTOP and WS_GROUP.

The default style for **RTEXT** is SS_RIGHT and WS_GROUP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

The following example demonstrates the use of the **RTEXT** statement:

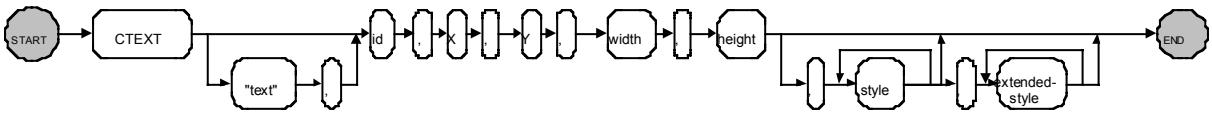
```
RTEXT "Number of Messages", 4, 30, 50, 100, 10
```

CTEXT

The **CTEXT** statement creates a centered-text control. The control is a simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line. The **CTEXT** statement, which you can use only in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of the control.

Syntax

```
CTEXT text, id, x, y, width, height [, style [,
extended-style]]
```



Parameters

text
: Specifies text that is centered in the rectangular area of the control.

Style
: Specifies the control styles. This value can be any combination of the following styles: SS_CENTER, WS_TABSTOP, and WS_GROUP.

If you do not specify a style, the default style is SS_CENTER and WS_GROUP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

This example creates a centered-text control that is labeled “filename”:

```
CTEXT "filename", 101, 10, 10, 100, 100
```

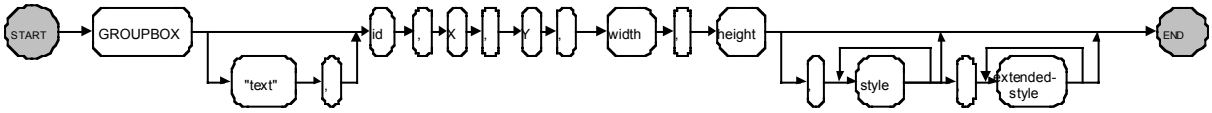
GROUPBOX

The **GROUPBOX** statement creates a group box control. The control is a rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given text in the upper-left corner. The **GROUPBOX** statement, which you can use only in a **DIALOG** statement, defines the text, identifier, dimensions, and attributes of a control window.

When the style contains WS_TABSTOP or the text specifies an accelerator, tabbing or pressing the accelerator key moves the focus to the first control within the group.

Syntax

```
GROUPBOX text, id, x, y, width, height [[, style
[[, extended-style]]]]
```



Parameters

style
: Specifies the control styles. This value can be a combination of the button class style BS_GROUPBOX and the WS_TABSTOP and WS_DISABLED styles.

If you do not specify a style, the default style is BS_GROUPBOX.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

This example creates a group-box control that is labeled “Options”:

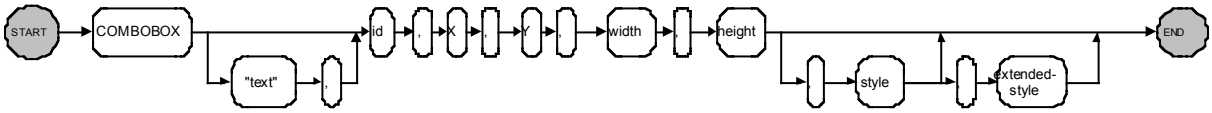
```
GROUPBOX "Options", 101, 10, 10, 100, 100
```

COMBOBOX

The **COMBOBOX** statement creates a combination box control (a combo box). A combo box consists of either a static text box or an edit box combined with a list box. The list box can be displayed at all times or pulled down by the user. If the combo box contains a static text box, the text box always displays the selection (if any) in the list box portion of the combo box. If it uses an edit box, the user can type in the desired selection; the list box highlights the first item (if any) that matches what the user has entered in the edit box. The user can then select the item highlighted in the list box to complete the choice. In addition, the combo box can be owner-drawn and of fixed or variable height.

Syntax

```
COMBOBOX text, id, x, y, width, height [, style [,
extended-style]]
```



Parameters

style
: Specifies the control styles. This value can be a combination of the COMBOBOX class styles and any of the following styles: WS_TABSTOP, WS_GROUP, WS_VSCROLL, and WS_DISABLED.

If you do not specify a style, the default style is CBS_SIMPLE and WS_TABSTOP.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

This example creates a combo-box control with a vertical scroll bar:

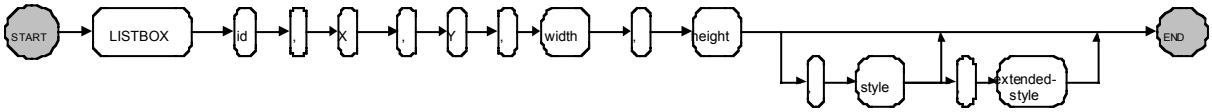
```
COMBOBOX 777, 10, 10, 50, 54, CBS_SIMPLE |
WS_VSCROLL | WS_TABSTOP
```

LISTBOX

The **LISTBOX** statement creates commonly used controls for a dialog box or window. The control is a rectangle containing a list of strings (such as filenames) from which the user can select. The **LISTBOX** statement, which can only be used in a **DIALOG** or **WINDOW** statement, defines the identifier, dimensions, and attributes of a control window.

Syntax

```
LISTBOX id, x, y, width, height [[, style [[,
extended-style]]]]
```



Parameters

style
: Specifies the control styles. This value can be a combination of the list-box class styles and any of the following styles: WS_BORDER and WS_VSCROLL.

If you do not specify a style, the default style is LBS_NOTIFY and WS_BORDER.

For more information on the *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

This example creates a list-box control whose identifier is 101:

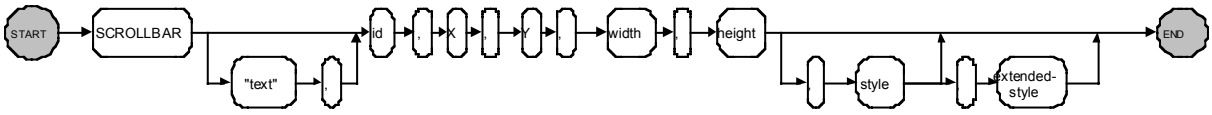
```
LISTBOX 101, 10, 10, 100, 100
```

SCROLLBAR

The **SCROLLBAR** statement creates a scroll-bar control. The control is a rectangle that contains a scroll box and has direction arrows at both ends. The scroll-bar control sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the scroll-box position. Scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input.

Syntax

```
SCROLLBAR text, id, x, y, width, height [[, style
[[, extended-style]]]]
```



Parameters

style
: Specifies a combination (or none) of the following styles: WS_TABSTOP, WS_GROUP, and WS_DISABLED.

In addition to these styles, the *style* parameter may contain a combination (or none) of the SCROLLBAR-class styles. The default style for **SCROLLBAR** is SBS_HORZ.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

Example

The following example demonstrates the use of the **SCROLLBAR** statement:

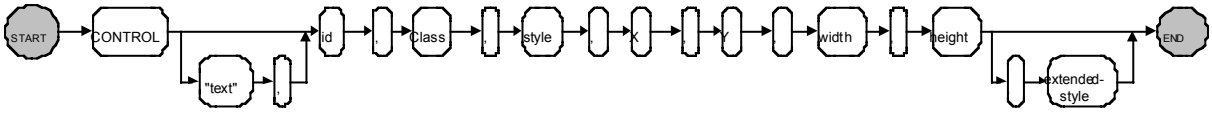
```
SCROLLBAR 999, 25, 30, 10, 100
```

CONTROL

This statement defines a user-defined control window.

Syntax

```
CONTROL text, id, class, style, x, y, width, height
[, extended-style]
```



Parameters

class
: Specifies a redefined name, character string, or a 16-bit unsigned integer value that defines the class. This can be any one of the control classes; for a list of the control classes, see the first list following this description. If the value is a redefined name supplied by the application, it must be a string enclosed in double quotation marks (**"**).

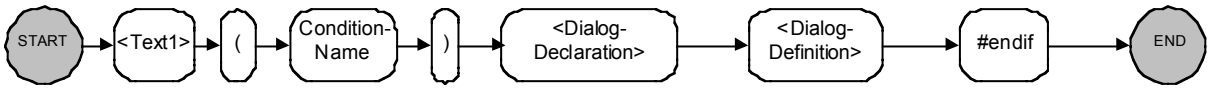
Style
: Specifies a redefined name or integer value that specifies the style of the given control. The exact meaning of *style* depends on the

class value. The sections following this description show the control classes and corresponding styles.

For more information on the *text*, *id*, *x*, *y*, *width*, *height*, *style*, and *extended-style* parameters, see Common Control Parameters.

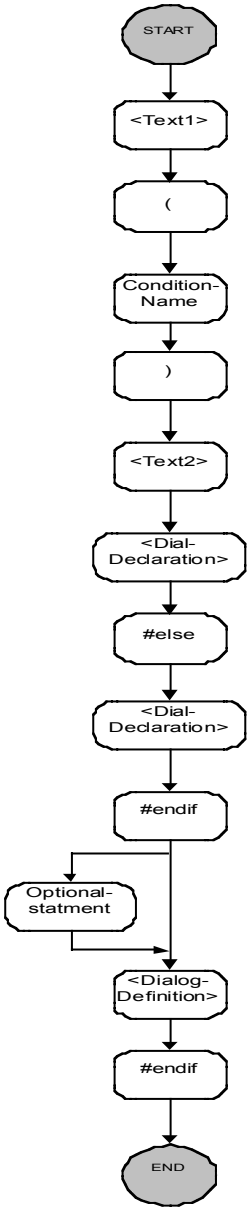
The six possible control classes are described in the following sections.

Copy Condition Declaration



Text 1

Condition Declaration



Text1


START
#if defined(APSTUDIO_INVOKED) || defined
END

Text2

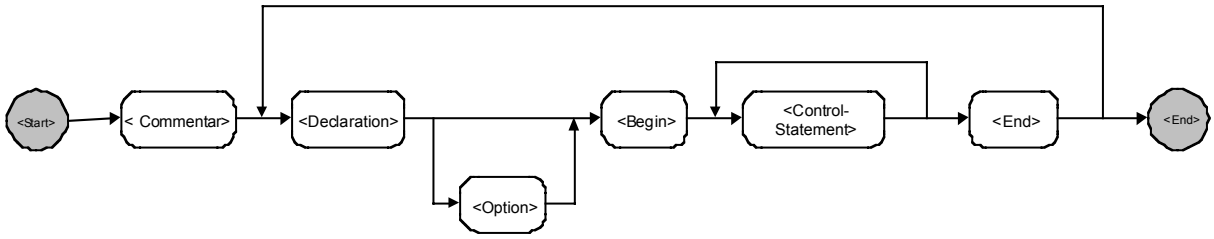

START
#if defined(APSTUDIO_INVOKED)
END

Syntax-Diagram Dialog

The **DIALOG** statement defines a window that an application can use to create dialog boxes. The statement defines the position and dimensions of the dialog box on the screen as well as the dialog box style.

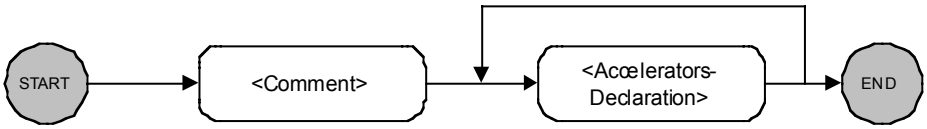
Syntax

```
nameID DIALOG [ load-mem] x, y, width, height
[optional-statements]
BEGIN
    control-statement
    . . .
END
```

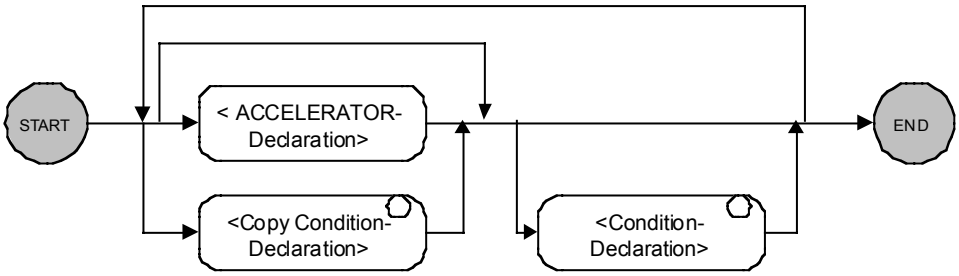


Syntax-Diagram Accelerators

The **ACCELERATORS** statement defines one or more accelerators for an application. An accelerator is a keystroke defined by the application to give the user a quick way to perform a task. The **TranslateAccelerator** function is used to translate accelerator messages from the application queue into WM_COMMAND or WM_SYSCOMMAND messages.

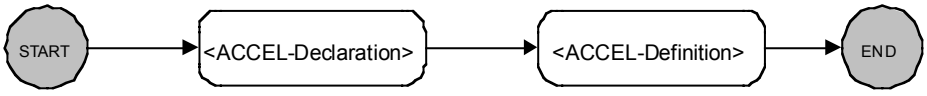


Accelerators Declaration

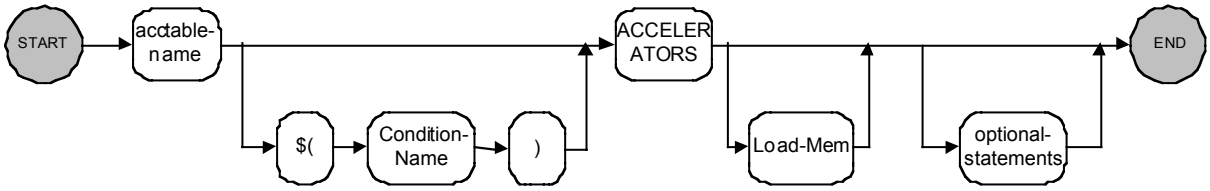


for each “Copy Condition-Declaration“ exist one “Condition-Declaration“ it Is also possible to have “Condition-Declaration“ if Accelerator is made with condition.

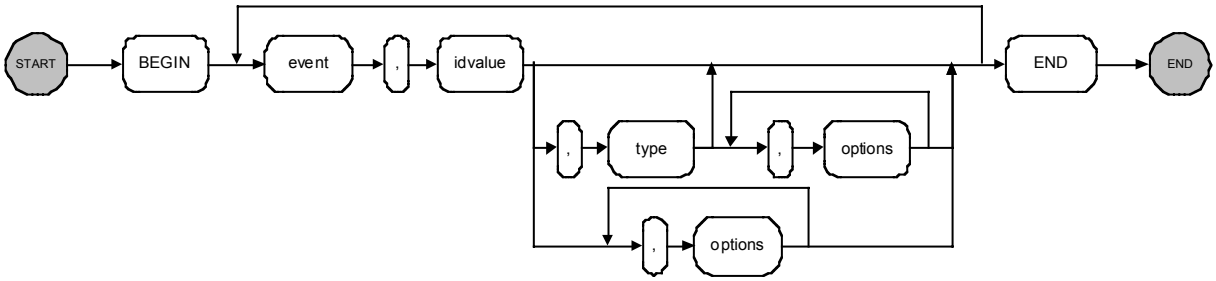
ACCELERATOR Declaration



ACCEL-Declaration



ACCEL-Definition



Syntax

```
acctablename ACCELERATORS
[optional-statements]
BEGIN
    event, idvalue, [type] [options]
    . . .
END
```

Parameters

acctablename
: Specifies either a unique name or a 16-bit unsigned integer value that identifies the resource.

optional-statements
: Zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that read and write resource files.

event

Specifies the keystroke to be used as an accelerator. It can be any one of the following character types:

"char"
: A single character enclosed in double quotation marks ("). The character can be preceded by a caret (^), meaning that the character is a control character.

Character
: An integer value representing a character. The type parameter must be ASCII.

virtual-key character
: An integer value representing a virtual key. The virtual key for alphanumeric keys can be specified by placing the uppercase letter or number in double quotation marks (for example, "9" or "C"). The type parameter must be VIRTKEY.

idvalue

Specifies a 16-bit unsigned integer value that identifies the accelerator.

type

Required only when the *event* parameter is a *character* or a *virtual-key character*. The *type* parameter specifies either **ASCII** or **VIRTKEY**; the integer value of *event* is interpreted accordingly. When **VIRTKEY** is specified and *event* contains a string, *event* must be uppercase.

options

Specifies the options that define the accelerator. This parameter can be one or more of the following values:

NOINVERT
: Specifies that no top-level menu item is highlighted when the accelerator is used. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item. If NOINVERT is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used.

ALT
: Causes the accelerator to be activated only if the ALT key is down.

SHIFT
: Causes the accelerator to be activated only if the SHIFT key is down.

CONTROL
: Defines the character as a control character (the accelerator is only activated if the CONTROL key is down). This has the same effect as using a caret (^) before the accelerator character in the *event* parameter.

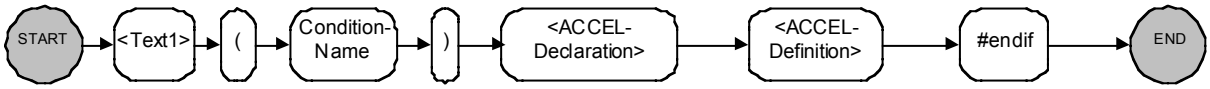
The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys.

Example

The following example demonstrates the use of accelerator keys:

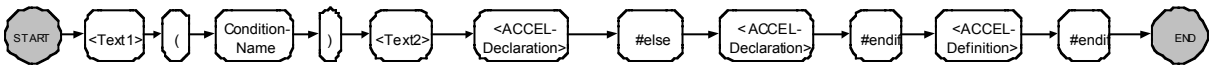
```
1 ACCELERATORS
BEGIN
  "^C",  IDDCLEAR         ; control C
  "K",   IDDCLEAR         ; shift K
  "k",   IDDELLIPSE, ALT  ; alt k
  98,    IDDRECT, ASCII   ; b
  66,    IDDSTAR, ASCII   ; B (shift b)
  "g",   IDDRECT          ; g
  "G",   IDDSTAR          ; G (shift G)
  VK_F1, IDDCLEAR, VIRTKEY                ; F1
  VK_F1, IDDSTAR, CONTROL, VIRTKEY        ; control F1
  VK_F1, IDDELLIPSE, SHIFT, VIRTKEY       ; shift F1
  VK_F1, IDDRECT, ALT, VIRTKEY            ; alt F1
  VK_F2, IDDCLEAR, ALT, SHIFT, VIRTKEY    ; alt shift F2
  VK_F2, IDDSTAR, CONTROL, SHIFT, VIRTKEY ; ctrl shift F2
  VK_F2, IDDRECT, ALT, CONTROL, VIRTKEY   ; alt control F2
END
```

Copy Condition Declaration



Text 1

Condition Declaration



Text1

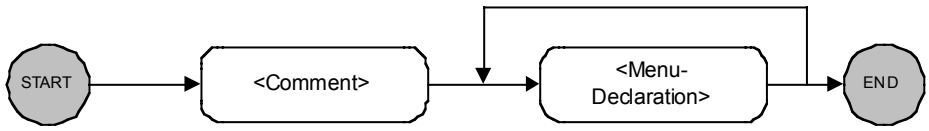
Text2

Syntax-Diagram MENU

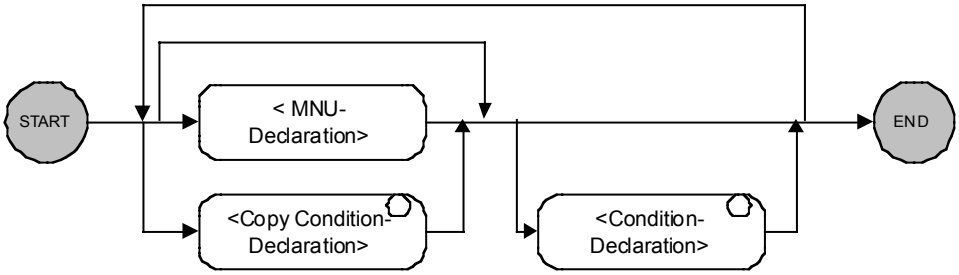
The **MENU** statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands from a list of command names.

Syntax

```
menuID MENU [load-mem]
[optional-statements]
BEGIN
    item-definitions
    . . .
END
```

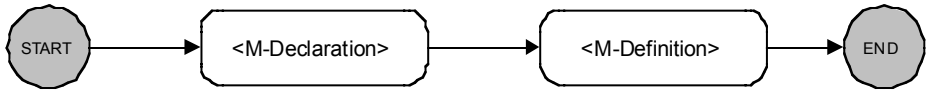


Menu Declaration

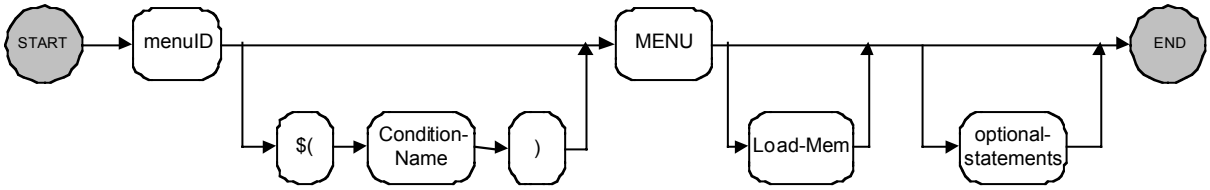


for each “Copy Condition-Declaration“ exist one “Condition-Declaration“ it is also possible to have “Condition-Declaration“ if Icon is made with condition.

MNU-Declaration



M-Declaration



menuID

Identifies the menu. This value is either a unique string or a unique 16-bit unsigned integer value in the range of 1 to 65,535.

load-mem

Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

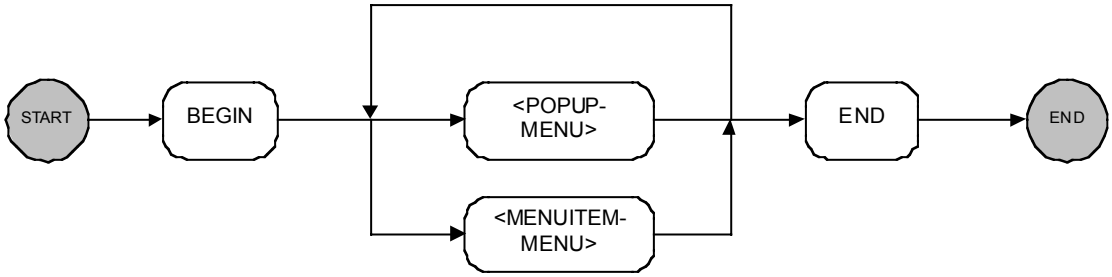
optional-statements

Zero or more of the following statements:

Statement	Description
CHARACTERISTICS *dword*	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE *language*, *sublanguage*	Specifies the language for the resource. The parameters are constants from WINNLS.H.
VERSION *dword*	User-defined version number for the resource that can be used by tools that

	read and write resource files.

M-Definition

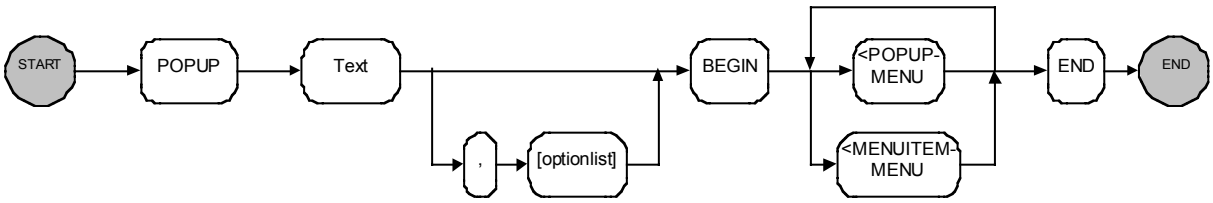


POPUP-MENU

The **POPUP** statement defines a menu item that can contain menu items and submenus.

```
Syntax
POPUP text, [[optionlist]]

BEGIN
   item-definitions
   .
   .
END
```



Parameters

text
: A string that contains the name of the menu. This string must be enclosed in double quotation marks (**"**).

Optionlist
: This parameter specifies redefined menu options that specify the appearance of the menu item. This optional parameter can be one or more of the following.

CHECKED
: Menu item has a check mark next to it. This option is not valid for a top-level menu.

GRAYED
: Menu item is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

INACTIVE
: Menu item is displayed but it cannot be selected.

MENUBARBREAK
: Same as **MENUBREAK** except that for menus, it separates the new column from the old column with a vertical line.

MENUBREAK
: Places the menu item on a new line for static menu-bar items. For menus, it places the menu item in a new column with no dividing line between the columns.

The **INACTIVE** and **GRAYED** options cannot be used together.

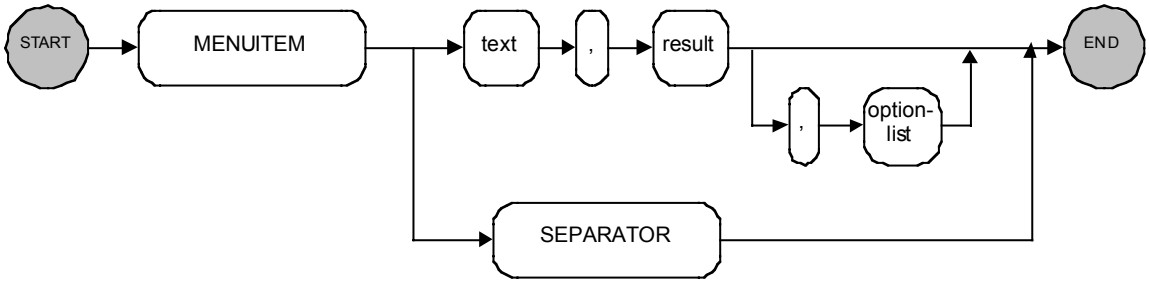
MENUITEM-MENU

MENUITEM

The **MENUITEM** statement defines a menu item.

Syntax

```
MENUITEM text, result, [optionlist]
MENUITEM SEPARATOR
```



Parameters

text
: Specifies the name of the menu item. The string can contain the escape characters \t and \a. The \t character inserts a tab in the string and is used to align text in columns. Tab characters should be used only in pop-up menus, not in menu bars. (For information on pop-up menus, see the POPUP statement.) The \a character aligns all text that follows it flush right to the menu bar or pop-up menu.

Result
: Specifies the result generated when the user selects the menu item. This parameter takes an integer value. Menu-item results are always integers; when the user clicks the menu-item name, the result is sent to the window that owns the menu.

Optionlist
: Specifies the appearance of the menu item. This optional parameter takes one or more redefined menu options, separated by commas or spaces. The menu options are as follows:

CHECKED
: Item has a check mark next to it.

GRAYED
: Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

HELP
: Identifies a help item.

INACTIVE
: Item name is displayed but it cannot be selected.

MENUBARBREAK
: Same as MF_MENUBREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.

MENUBREAK
: Places the menu item on a new line for static menu-bar items. For pop-up menus, it places the menu item in a new column with no

dividing line between the columns. The INACTIVE and GRAYED options cannot be used together.

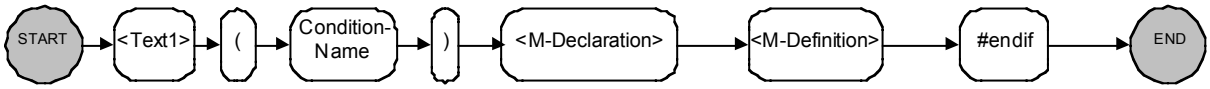
SEPARATOR
The MENUITEM SEPARATOR form of the MENUITEM statement creates an inactive menu item that serves as a dividing bar between two active menu items in a pop-up menu.

Example

The following example demonstrates the use of the **MENUITEM** and **MENUITEM SEPARATOR** statements:

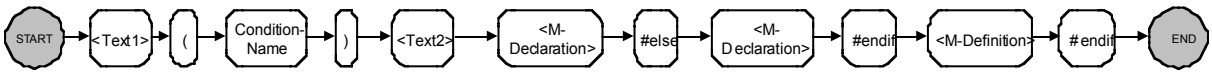
```
MENUITEM "&Roman", 206, CHECKED, GRAYED
MENUITEM SEPARATOR
MENUITEM "&Blackletter", 301
```

Copy Condition Declaration



Text 1

Condition Declaration

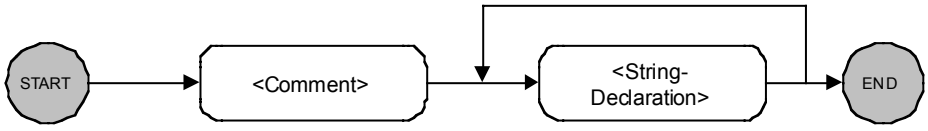


Text1

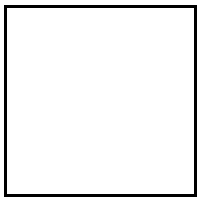
Text2

Syntax-Diagram String Table

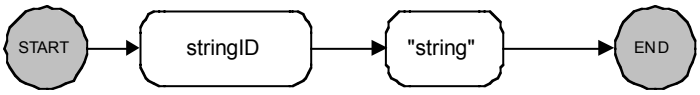
The **STRINGTABLE** statement defines one or more string resources for an application. String resources are simply null-terminated Unicode strings that can be loaded when needed from the executable file, using the **LoadString** function.



String-Declaration



String-Definition



Syntax

```
STRINGTABLE [[load-mem]]
[[optional-statements]]
BEGIN
    stringID string
    . . .
END
```

Parameter

load-mem
: Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

optional-statements
: Zero or more of the following statements:

Statement	Description
CHARACTERISTICS dword	User-defined information about a resource that can be used by tools that read and write resource files.
LANGUAGE language, sublanguage	Specifies the language for the resource. The parameters are constants from WINNT.H.
VERSION dword	User-defined version number for the resource that can be used by tools that read and write resource files.

StringID
: Specifies an unsigned 16-bit integer that identifies the resource.

String
: Specifies one or more strings, enclosed in double quotation marks. The string must be no longer than 4097 characters and must occupy a single line in the source file. To add a carriage return to the string, use this character sequence: **\012**. For example, "Line one\012Line two" would define a string that would be displayed as follows: `Line one Line two`

Remarks

Grouping strings in separate sections allows all related strings to be read in at one time and discarded together. When possible, an application should make the table movable and discardable. RC allocates 16 strings per section and uses the identifier value to determine which section is to contain the string. Strings with the same upper-12 bits in their identifiers are placed in the same section.

Example

The following example demonstrates the use of the **STRINGTABLE** statement:

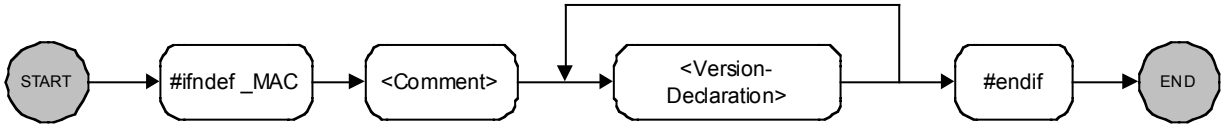
```
#define IDSHELLO    1
#define IDSGOODBYE  2

STRINGTABLE
BEGIN
    IDSHELLO,   "Hello"
    IDSGOODBYE, "Goodbye"
END
```

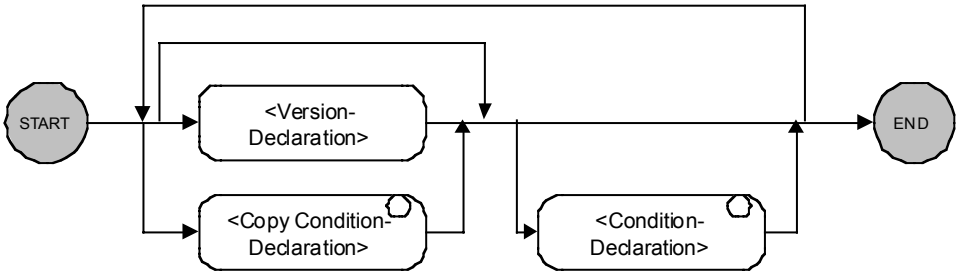
Syntax-Diagram Versioninfo

The **VERSIONINFO** statement creates a version-information resource. The resource contains such information about the file as its version number, its intended operating system, and its original filename. The resource is intended to be used with the File Installation library functions.

```
Syntax
versionID VERSIONINFO fixed-info
BEGIN
   block-statement
   ...
END
```



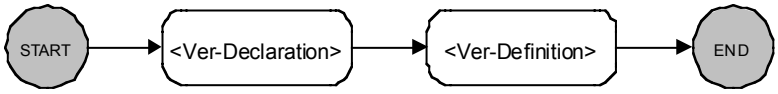
Version Declaration



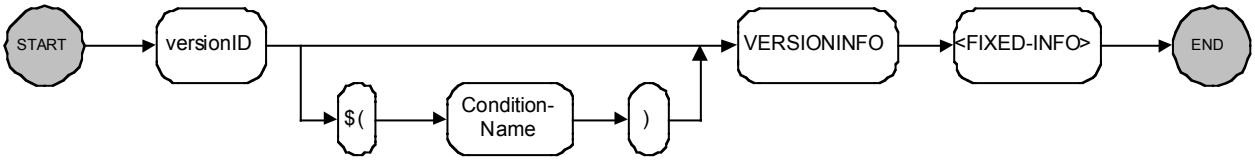
for each “Copy Condition-Declaration“ exist one “Condition-Declaration“

it Is also possible to have “Condition-Declaration“ if Version is made with condition

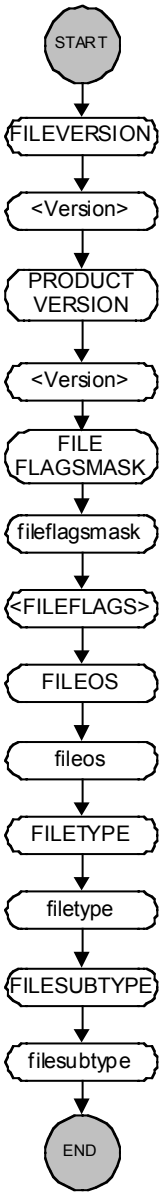
Version-Declaration



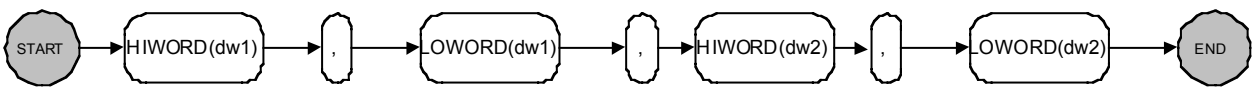
Ver-Declaration



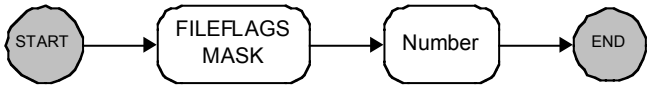
FIXED-INFO



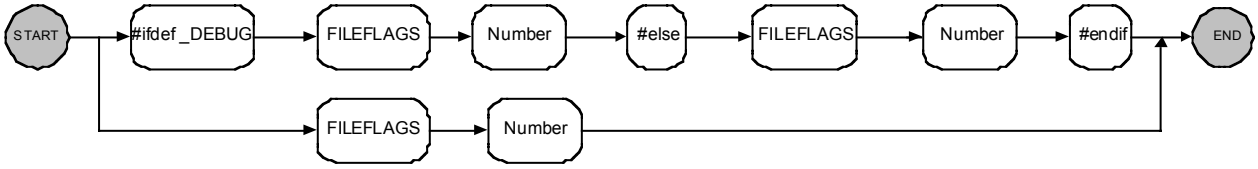
Version



fileflagsmask



FILEFLAGS



Parameters

versionID
: Specifies the version-information resource identifier. This value must be 1.

fixed-info
: Specifies the version information, such as the file version and the intended operating system. This parameter consists of the following statements:

FILEVERSION version
: Specifies the binary version number for the file. The version consists of two 32-bit integers, defined by four 16-bit integers. For example, “FILEVERSION 3,10,0,61” is translated into two doublewords: 0x0003000a and 0x0000003d, in that order. Therefore, if version is defined by the doublewords dw1 and dw2, they need to appear in the **FILEVERSION** statement as follows: **HIWORD**(dw1), **LOWORD**(dw1), **HIWORD**(dw2), **LOWORD**(dw2).

PRODUCTVERSION version
: Specifies the binary version number for the product with which the file is distributed. The version parameter is two 32-bit integers, defined by four 16-bit integers. For more information about version, see the **FILEVERSION** description.

FILEFLAGSMASK fileflagsmask
: Specifies which bits in the **FILEFLAGS** statement are valid. If a bit is set, the corresponding bit in **FILEFLAGS** is valid.

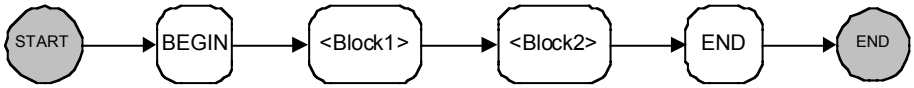
FILEFLAGS fileflags
: Specifies the Boolean attributes of the file. The fileflags parameter must be the combination of all the file flags that are valid at compile time. For Windows 3.1, this value is 0x3f.

FILEOS fileos
: Specifies the operating system for which this file was designed. The fileos parameter can be one of the operating system values given in the Comments section.

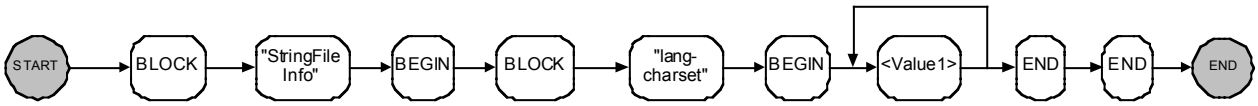
FILETYPE filetype
: Specifies the general type of file. The filetype parameter can be one of the file type values listed in the Comments section.

FILESUBTYPE subtype
: Specifies the function of the file. The subtype parameter is zero unless the type parameter in the **FILETYPE** statement is **VFT_DRV**, **VFT_FONT**, or **VFT_VXD**. For a list of file subtype values, see the Comments section.

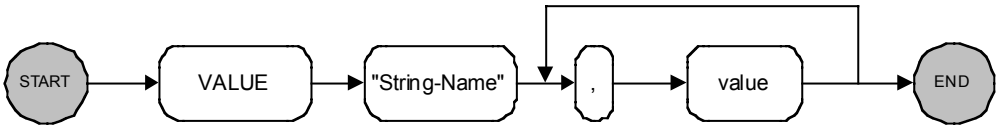
Ver-Definition



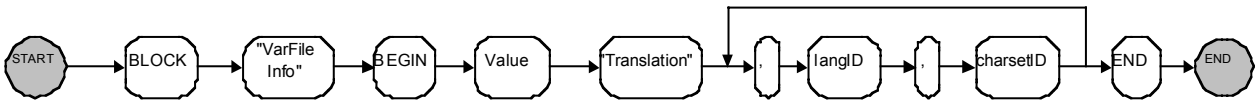
Block1



Value1



Block2



A string information block has the following form:

```
BLOCK "StringFileInfo"
BEGIN
    BLOCK "lang-charset"
    BEGIN
        VALUE "string-name", "value"
            . . .
    END
END
```

Following are the parameters in the **StringFileInfo** block:

lang-charset
: Specifies a language and character-set identifier pair. It is a hexadecimal string consisting of the concatenation of the language and character-set identifiers listed earlier in this section.

string-name
: Specifies the name of a value in the block and can be one of the redefined names listed earlier in this section.

Value
: Specifies, as a character string, the value of the corresponding string name. More than one **VALUE** statement can be given.

A variable information block has the following form:

```
BLOCK "VarFileInfo"
BEGIN
    VALUE "Translation",
        langID, charsetID
            . . .
END
```

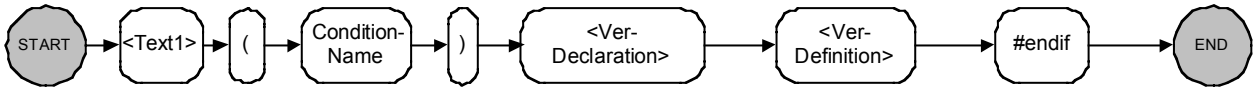
Following are the parameters in the variable information block:

langID
: Specifies one of the language identifiers listed earlier in this section.

CharsetID

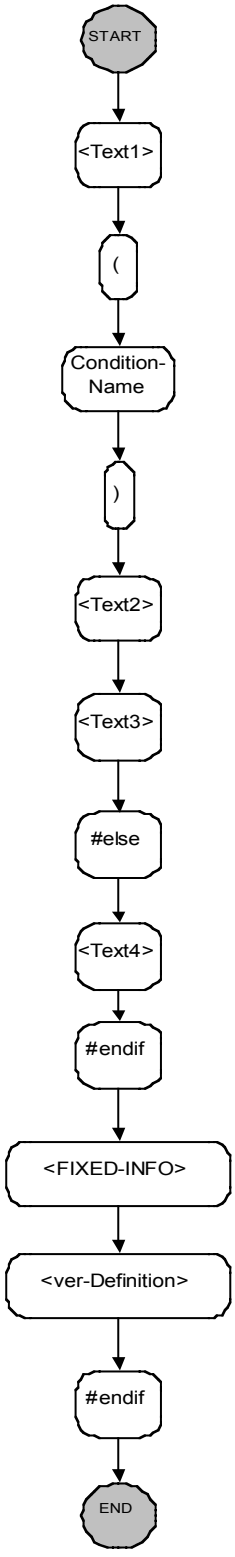
Specifies one of the character-set identifiers listed earlier in this section. More than one identifier pair can be given, but each pair must be separated from the preceding pair with a comma.

Copy Condition Declaration



Text 1

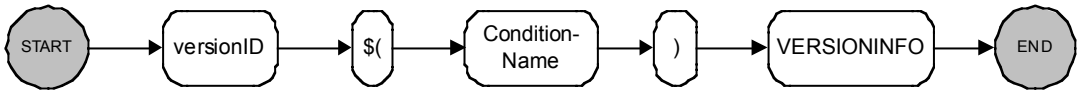
Condition Declaration



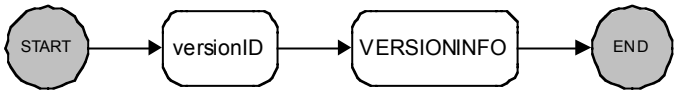
Text1

Text2

Text3



Text4



block-statement

Specifies one or more version-information blocks. A block can contain string information or variable information.

Remarks

To use the constants specified with the **VERSIONINFO** statement, the WINVER.H file (included in WINDOWS.H) must be included in the resource-definition file.

The following list describes the parameters used in the **VERSIONINFO** statement:

fileflags
Specifies a combination of the following values:

VS_FF_DEBUG
File contains debugging information or is compiled with debugging features enabled.

VS_FF_INFOINFERRED
File contains a dynamically created version-information resource. Some of the blocks for the resource may be empty or incorrect. This value is not intended to be used in version-information resources created by using the **VERSIONINFO** statement.

VS_FF_PATCHED
File has been modified and is not identical to the original shipping file of the same version number.

VS_FF_PRERELEASE
File is a development version, not a commercially released product.

VS_FF_PRIVATEBUILD
File was not built using standard release procedures. If this value is given, the **StringFileInfo** block must contain a **PrivateBuild** string.

VS_FF_SPECIALBUILD
File was built by the original company using standard release procedures but is a variation of the standard file of the same version number. If this value is given, the **StringFileInfo** block must contain a **SpecialBuild** string.

Fileos
Specifies one of the following values:

Value	Description
VOS_UNKNOWN	Operating system for which the file was

	designed is unknown to Windows.
VOS_DOS	File was designed for MS-DOS.
VOS_NT	File was designed for Windows NT.
VOS_WINDOWS16	File was designed for Windows version 3.0 or later.
VOS_WINDOWS32	File was designed for 32-bit Windows.
VOS_DOS_WINDOWS16	File was designed for Windows version 3.0 or later running with MS-DOS.
VOS_DOS_WINDOWS32	File was designed for 32-bit Windows running with MS-DOS.
VOS_NT_WINDOWS32	File was designed for 32-bit Windows running with Windows NT.

The values 0x00002L, 0x00003L, 0x20000L and 0x30000L are reserved.

Filetype

Specifies one of the following values:

Value	Description
VFT_UNKNOWN	File type is unknown to Windows.
VFT_APP	File contains an application.
VFT_DLL	File contains a dynamic-link library (DLL).
VFT_DRV	File contains a device driver. If the dwFileType member is VFT_DRV, the dwFileSubtype member contains a more specific description of the driver.
VFT_FONT	File contains a font. If the dwFileType member is VFT_FONT, the dwFileSubtype member contains a more specific description of the font.
VFT_VXD	File contains a virtual device.
VFT_STATIC_LIB	File contains a static-link library.

All other values are reserved for use by Microsoft.

subtype

Specifies additional information about the file type.

If the **FILETYPE** statement specifies **VFT_DRV**, this parameter can be one of the following values:

Value	Description
VFT2_UNKNOWN	Driver type is unknown to Windows.
VFT2_DRV_COMM	File contains a communications driver.
VFT2_DRV_PRINTER	File contains a printer driver.
VFT2_DRV_KEYBOARD	File contains a keyboard driver.
VFT2_DRV_LANGUAGE	File contains a language driver.
VFT2_DRV_DISPLAY	File contains a display driver.
VFT2_DRV_MOUSE	File contains a mouse driver.
VFT2_DRV_NETWORK	File contains a network driver.
VFT2_DRV_SYSTEM	File contains a system driver.
VFT2_DRV_INSTALLABLE	File contains an installable driver.
VFT2_DRV_SOUND	File contains a sound driver.

If the **FILETYPE** statement specifies **VFT_FONT**, this parameter can be one of the following values:

Value	Description
VFT2_UNKNOWN	Font type is unknown to Windows.
VFT2_FONT_RASTER	File contains a raster font.
VFT2_FONT_VECTOR	File contains a vector font.
VFT2_FONT_TRUETYPE	File contains a TrueType font.

If the **FILETYPE** statement specifies **VFT_VXD**, this parameter must be the virtual-device identifier included in the virtual-device control block.

All *subtype* values not listed here are reserved for use by Microsoft.

LangID

Specifies one of the following language codes:

Code	Language	Code	Language
0x0401	Arabic	0x0415	Polish
0x0402	Bulgarian	0x0416	Brazilian Portuguese
0x0403	Catalan	0x0417	Rhaeto-Romanic
0x0404	Traditional Chinese	0x0418	Romanian
0x0405	Czech	0x0419	Russian
0x0406	Danish	0x041A	Croato-Serbian (Latin)
0x0407	German	0x041B	Slovak
0x0408	Greek	0x041C	Albanian
0x0409	U.S. English	0x041D	Swedish
0x040A	Castilian Spanish	0x041E	Thai
0x040B	Finnish	0x041F	Turkish
0x040C	French	0x0420	Urdu
0x040D	Hebrew	0x0421	Bahasa
0x040E	Hungarian	0x0804	Simplified Chinese
0x040F	Icelandic	0x0807	Swiss German
0x0410	Italian	0x0809	U.K. English
0x0411	Japanese	0x080A	Mexican Spanish
0x0412	Korean	0x080C	Belgian French
0x0413	Dutch	0x0C0C	Canadian French
0x0414	Norwegian – Bokml	0x100C	Swiss French
0x0810	Swiss Italian	0x0816	Portuguese
0x0813	Belgian Dutch	0x081A	Serbo-Croatian (Cyrillic)
0x0814	Norwegian Nynorsk		

charsetID

Specifies one of the following character-set identifiers:

Identifier	Character Set
0	7-bit ASCII
932	Windows, Japan (Shift – JIS X-0208)
949	Windows, Korea (Shift – KSC 5601)
950	Windows, Taiwan (GB5)
1200	Unicode
1250	Windows, Latin-2 (Eastern European)
1251	Windows, Cyrillic
1252	Windows, Multilingual
1253	Windows, Greek
1254	Windows, Turkish
1255	Windows, Hebrew

1256	Windows, Arabic

string-name
Specifies one of the following redefined names:

Comments
Specifies additional information that should be displayed for diagnostic purposes.

CompanyName
Specifies the company that produced the file—for example, “Microsoft Corporation” or "Standard Microsystems Corporation, Inc.” This string is required.

FileDescription
Specifies a file description to be presented to users. This string may be displayed in a list box when the user is choosing files to install—for example, “Keyboard Driver for AT-Style Keyboards” or "Microsoft Word for Windows”. This string is required.

FileVersion
Specifies the version number of the file—for example, “3.10” or “5.00.RC2”. This string is required.

InternalName
Specifies the internal name of the file, if one exists—for example, a module name if the file is a dynamic-link library. If the file has no internal name, this string should be the original filename, without extension. This string is required.

LegalCopyright
Specifies all copyright notices that apply to the file. This should include the full text of all notices, legal symbols, copyright dates, and so on—for example, “Copyright© Microsoft Corporation 1990–1992”. This string is optional.

LegalTrademarks
Specifies all trademarks and registered trademarks that apply to the file. This should include the full text of all notices, legal symbols, trademark numbers, and so on—for example, “Windows™ is a trademark of Microsoft® Corporation”. This string is optional.

OriginalFilename
Specifies the original name of the file, not including a path. This information enables an application to determine whether a file has been renamed by a user. The format of the name depends on the file system for which the file was created. This string is required.

PrivateBuild
Specifies information about a private version of the file—for example, “Built by TESTER1 on \TESTBED”. This string should be present only if the **VS_FF_PRIVATEBUILD** flag is set in the **dwFileFlags** member of the **VS_FIXEDFILEINFO** structure of the root block.

ProductName
Specifies the name of the product with which the file is distributed—for example, “Microsoft Windows”. This string is required.

ProductVersion
Specifies the version of the product with which the file is distributed—for example, “3.10” or “5.00.RC2”. This string is required.

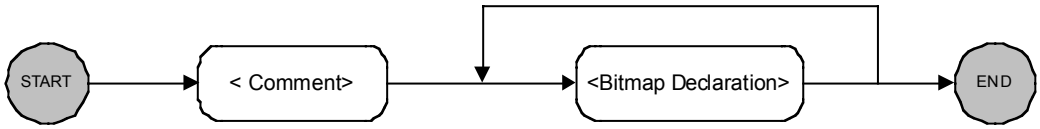
SpecialBuild
Specifies how this version of the file differs from the standard

version—for example, “Private build for TESTER1 solving mouse problems on M250 and M250E computers”. This string should be present only if the **VS_FF_SPECIALBUILD** flag is set in the **dwFileFlags** member of the **VS_FIXEDFILEINFO** structure in the root block.

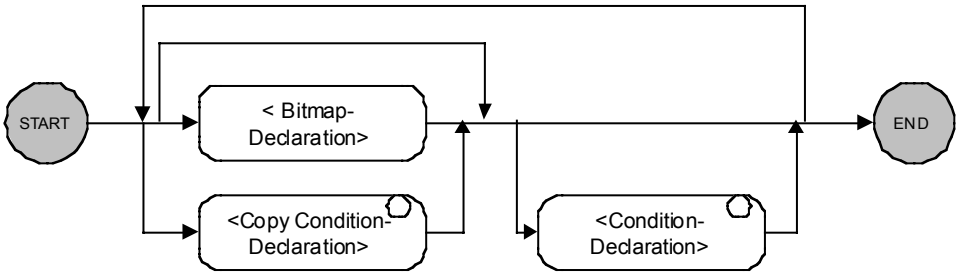
Syntax-Diagram Bitmap

BITMAP

The **BITMAP** resource-definition statement specifies a bitmap that an application uses in its screen display or as an item in a menu or control.

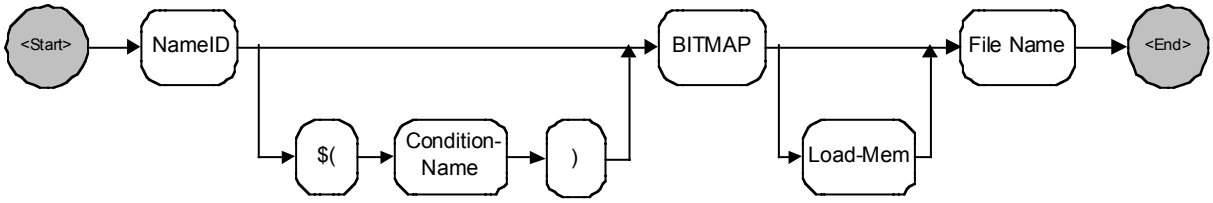


Bitmap Declaration



for each “Copy Condition-Declaration“ exist one “Condition-Declaration“ it is also possible to have “Condition-Declaration“ if Icon is made with condition.

Declaration



Syntax

```
nameID BITMAP [load-mem] filename
```

Parameter

nameID
: Specifies either a unique name or a 16-bit unsigned integer value identifying the resource.

load-mem
: Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

Filename
: Specifies the name of the file that contains the resource. The name

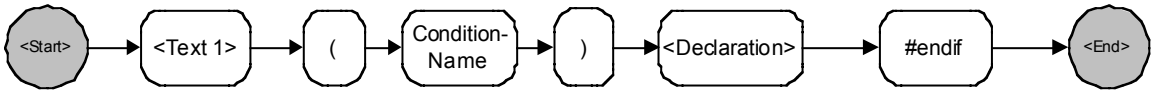
must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Example

The following example specifies two bitmap resources:

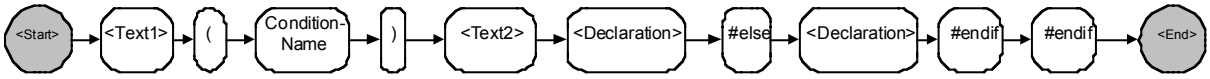
```
disk1   BITMAP disk.bmp
12      BITMAP PRELOAD diskette.bmp
```

Condition Declaration



Text 1

End Condition Declaration

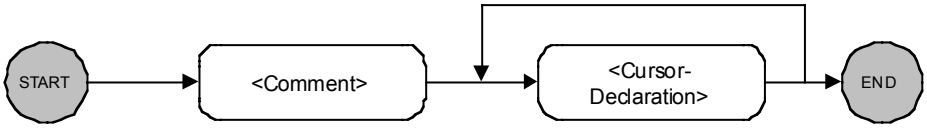


Text1

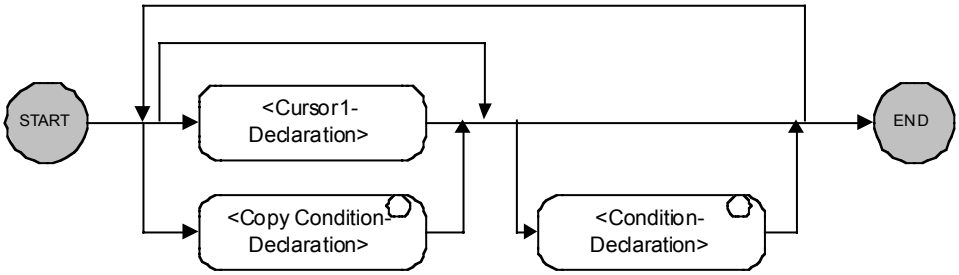
Text2

Syntax-Diagram Cursor

The **CURSOR** statement specifies a bitmap that defines the shape of the cursor on the display screen.

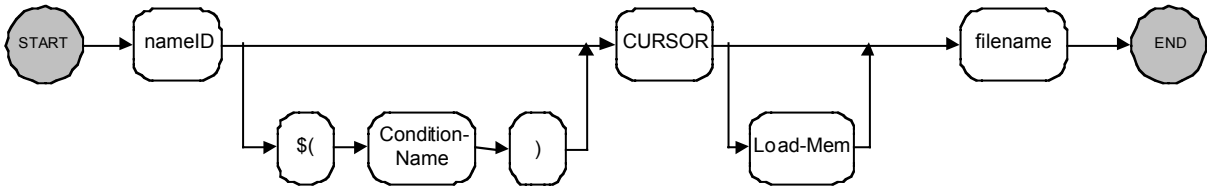


Cursor Declaration



for each “Copy Condition-Declaration“ exist one “Condition-Declaration“ it is also possible to have “Condition-Declaration“ if Bitmap is made with condition.

Cursor1-Declaration



Syntax

```
nameID CURSOR [load-mem] filename
```

Parameters

nameID
: Specifies either a unique name or a 16-bit unsigned integer identifying the resource.

load-mem
: Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

Filename
: Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Remarks

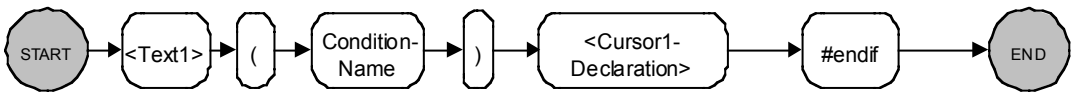
Icon and cursor resources can contain more than one image. If the resource is marked with the **PRELOAD** option, Windows loads all images in the resource when the application executes.

Example

The following example specifies two cursor resources; one by name (cursor1) and the other by number (2):

```
cursor1 CURSOR bullseye.cur
2       CURSOR "d:\\cursor\\arrow.cur"
```

Copy Condition Declaration



Text 1

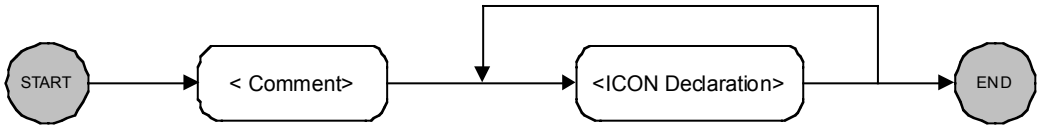
Condition Declaration

Text1

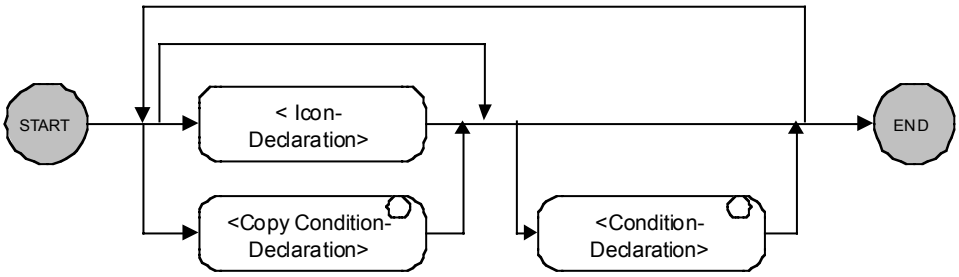
Text2

Syntax-Diagram Icon

The **ICON** resource-definition statement specifies a bitmap that defines the shape of the icon to be used for a given application.

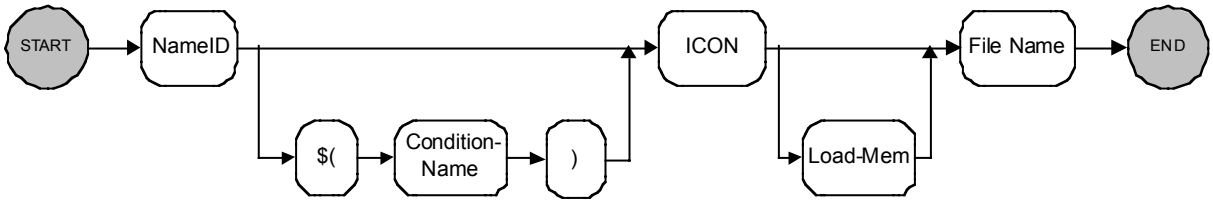


ICON Declaration



for each “Copy Condition-Declaration“ exist one “Condition-Declaration“ it is also possible to have “Condition-Declaration“ if Icon is made with condition.

Declaration



Syntax

```
nameID ICON [load-mem] filename
```

Parameters

nameID
: Specifies either a unique name or a 16-bit unsigned integer value identifying the resource.

load-mem
: Specifies loading and memory attributes for the resource. For more information, see Common Resource Attributes.

Filename
: Specifies the name of the file that contains the resource. The name must be a valid filename; it must be a full path if the file is not in the current working directory. The path can either be a quoted or nonquoted string.

Remarks

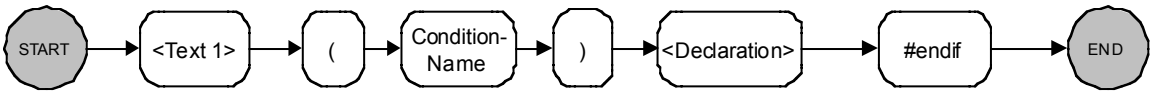
Icon and cursor resources can contain more than one image. If the resource is marked as **PRELOAD**, Windows loads all images in the resource when the application executes.

Example

The following example specifies two icon resources:

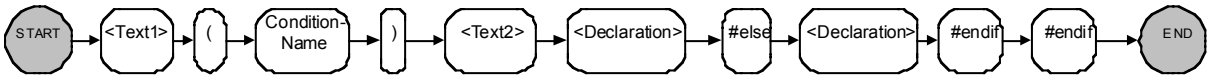
```
desk1   ICON desk.ico
11      ICON DISCARDABLE custom.ico
```

Condition Declaration



Text 1

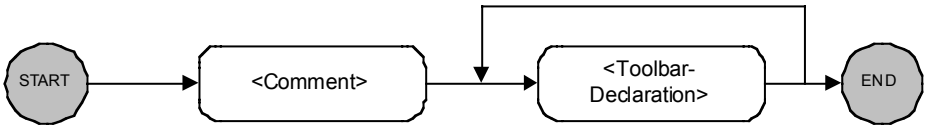
End Condition Declaration



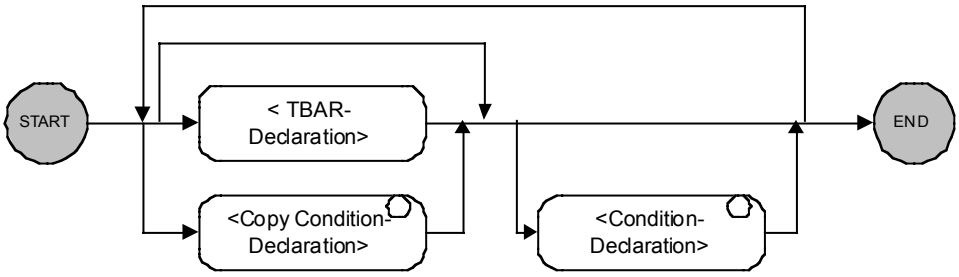
Text1

Text2

Syntax-Diagram Toolbar

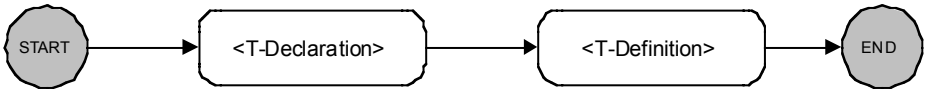


Toolbar Declaration

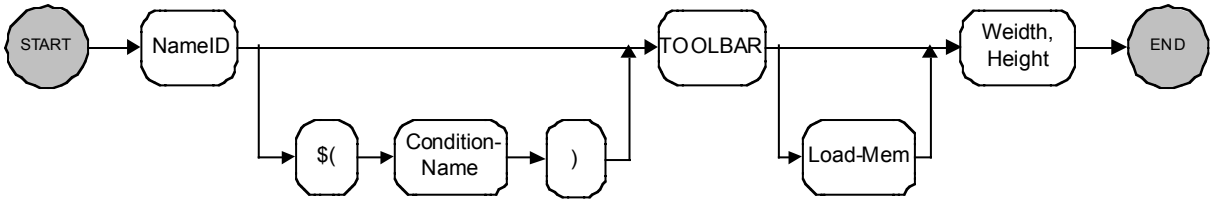


for each “Condition-Declaration“ exsist one “End Condition-Declaration“ but is not defined where

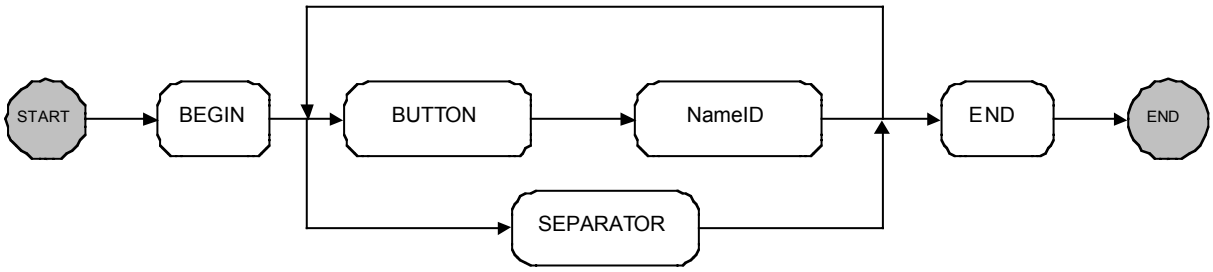
TBAR-Declaration



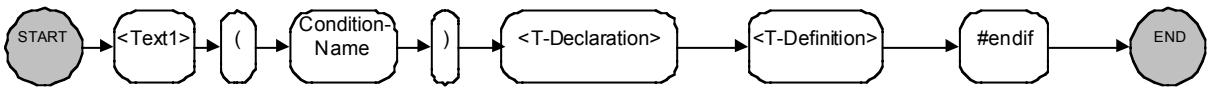
T-Declaration



T-Definition

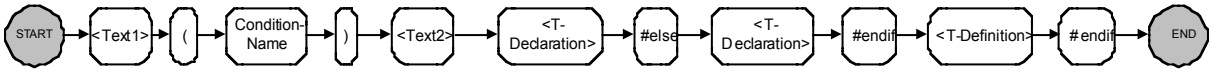


Copy Condition Declaration



Text 1

Condition Declaration



Text1

Text2

Glossar

Index

Error! No index entries found.

ResourceWizard Designentscheide

Vorbemerkungen

Zuerst möchte ich etwas über meine Erfahrungen als Programmierer berichten, bevor ich bei der TeTrade AG diesen Werkvertrag unterzeichnete. Ich hatte bereits zwei Jahre bei der B&S Ingenieure und Planer AG als Entwickler vorwiegend für Datenbankanwendungen unter Windows gearbeitet. Die dortige Entwicklungsumgebung war und ist Delphi. Somit brachte ich etwas Know-How insbesondere im Bezug zum Windows-API und GUI-Programmierung mit, weshalb ich mich nicht auf eine Umstellung der Programmiersprache einerseits (von Objective-Pascal nach C++) und der Umgebung andererseits (MS Visual C++) scheute, zumal ich die meisten Implementierungsaufgaben im Rahmen der Informatiklesungen meistens in C++ verfasste.
Dass ich das Projekt nicht bei der B&S AG absolvierte, lag daran, dass dort keine Aufgabe im vorgegebenen Zeitrahmen anstand. Ausserdem war der Projektbetreuer Andreas Münger ein vertrauter Kommiliton.

Designentscheide

Die folgenden Abschnitte sind entsprechend den verschiedenen Teilaufgaben vom Resourcewizard geordnet, wie Sie sie auch im Design-Dokument finden können.

Grobe interne Struktur

Nachdem die Anforderungen bekannt waren, ging es vorerst um eine grobe Aufteilung der Aufgaben. Diese sind offensichtlich: Interne Repräsentation der Daten, Parsen, GUI und Datenbankanbindung für den Dictionär.

Interne Repräsentation der Daten

Resourcen werden im MS Visual Studio mittels eines Treeviews dargestell, in der ersten Ebene die Kategorien, in der zweiten Instanzen (z.B: Dialoge->OpenDialog). Eine Aufteilung anhand der Kategorien und Resourcentypen in einzelne Klassen schien mir deshalb sinnvoll. Ausserdem lag eine Baumstruktur auf der Hand. Die Idee eines CTreeItem, das die meisten Funktionalitäten virtuell oder gar abstrakt zur Verfügung, stellt war geschaffen. Alle weiteren Resourcentypen oder Kategorien sollten Derivate von dieser Klasse bilden. Da die MFC-Bibliothek (=Microsoft Foundation Classes,die mit Visual Studio mitgelieferte Klassenbibliothek) keinen Baum-Container zur Verfügung stellt und die Integration der STL (Standard Template Library) fehlschlug, wurden die für eine Baumstruktur nötigen Funktionen in CTreeItem eingebettet, also z.B. „Wer ist Vater-Objekt“, Iteration über alle Söhne, etc.

Als nächstes ging es darum zu überlegen, wie die Mehrsprachenlogik einzubinden sein sollte. Da sich zwei Resourcen die gleichen Sprachinhalte teilen können, war hier eine direkte Einbindung in CTreeItem nicht sinnvoll. Ein CTreeItem sollte also lediglich eine Referenz auf einen solchen Sprachinhalt enthalten. Das so geschaffene CTextItem seinerseits weiss im Gegenzug, von welchen CTreeItems es „benutzt“ wird. Weitere Anforderungen der Applikation rechtfertigen diesen Entscheid: CTextItems können verschmolzen (zwei CTreeItem haben nach einer Änderung die selben Sprachinhalte), aufgeteilt (zwei CTreeItems, die dasselbe CTextItem referenzieren, sollen nun verschiedene Sprachinhalte bekommen), gelöscht oder geschaffen werden. Dieser Ansatz vereinfacht auch die spätere GUI-Aufbereitung als Tabellengitter.

Eine Resourcendatei repräsentiert sich also mittels CTreeItem-Derivaten, die baumartig verknüpft sind und einer Liste von CTextItems. Zur einfachen Verwaltung entschied ich mich zu zwei speziellen Klassen CResourceTree (das Rootelement des Baumes) und CTextTable (verantwortlich für die CTextItems). Dieses Paar bildet das „Dokument“ (s. Kapitel GUI).

Parsen

Eine Einbindung in CTreeItem schien mir ungeeignet: Der Parser hat die Aufgabe, anhand der Resourcendatei CTreeItem- und CTextItem- Instanzen zu schaffen. Ein „externer“ Parser sollte es also sein.
Ich hatte kaum Erfahrung mit Parsen - ich hatte gerade erst die Lesung Compilerbau hinter mich gebracht. Auch in der TeTrade schien das Thema mehr oder weniger Neuland zu sein - zumindest auf C++-Ebene. Leider konnte ich mich trotz der Compilerbaulesung nicht mit lex und yacc anfreunden - obwohl dort offensichtlich viele Probleme schon gelöst waren. Mir schien eine Einbettung in eine objektorientierte Architektur doch etwas umständlich. Im Rahmen der Lesung hatte ich allerdings (in purem C) eine „allgemeine“ Parserlösung zustandegebracht. Ein Umschreiben auf C++ erschien einfach und sinnvoll, zumal ein solcher Parser auch für andere Aufgaben einsetzbar wäre.

GUI

Die MFC-Bibliothek gibt bereits grob eine Dokument-View-Struktur vor: CDocument für die Kapselung der Daten (in diesem Fall die Resourcendatei), CView für eine allgeime GUI-Aufbereitung derselben. Die Projektschablonen der Entwicklungsumgebung generieren denn auch entsprechende Derivate. Zudem stehen für Standard-Windows-Controls entsprechende View-Klassen bereit (beispielsweise CTreeView für ein Treecontrol). Mehrere Views Teilen sich ein sogenanntes FrameWindow, ein Fenster.

Für die Baumansicht war ein CTreeView offensichtlich. Die Tabellenansicht mittels eines CListView zu realisieren schien aber unmöglich. Ein CListView ist beispielsweise die rechte Seite im Win32-Datei-Explorer. Dieses Kontrollelement lässt zwar Icons und mehrere Spalten zu, doch kann man beispielsweise nur erste Spalte editieren. Auch weitere Probleme liess mich dieses Control als Lösung verwerfen. Leider enthält die MFC-Bibliothek nicht gerade viele brauchbare GUI-Wrapper. MFC bildet gerade mal die Standardelemente wie Checkboxen, Liste, Eingabe- oder Auswahlfeld ab.

Ein Blick in den Sourcecode eines entsprechenden Elementes in der Delphi-Bibliothek brachte mich auch schnell von der Idee ab, eine solches Element selber zu erschaffen. Ein solches Vorhaben wäre zu aufwendig und ungerechtfertigt gewesen. In Absprache mit B. Burkhardt entschied ich mich für den Einkauf einer „Komponente“: Objective-Grid von Stingray. Diese Klassenbibliothek lässt sich einfach in Visual C++ einbinden und stellt im wesentlichen eine GUI-Tabellenfunktionalität in einem Excel-Look-and-Feel zur Verfügung.

Die später hinzugekommenen Anforderungen liessen sich einigermassen einfach einbetten: Für die Referenzansicht eignete sich nun ein CListView, für die Vorschau ein einfacher CView, da die Elemente ohnehin „von Hand“ gezeichnet werden mussten.

Datenbank

Die Ansteuerung von MS-Access-Datenbanken ist mit entsprechenden MFC-Klassen einfach, deshalb kam eine andere Datenbank nicht in Frage. Da der Dictionär ein schnelles Nachschlagen von Begriffen erlauben sollte, wurde auf eine komplexe Struktur verzichtet. Eine redundante Lösung mit nur einer Tabelle schien mir deshalb sinnvoll. Für noch schnelleres Nachschlagen lassen sich die Daten mittels einer CMap im Speicher halten. Dies ist eine MFC-Template-Klasse mit einer Hash-Funktionalität.

Schlussbemerkungen

In diesem Projekt machte ich erstmals mit dem wahrscheinlichen Hauptproblem in der Informatik Erfahrung: Eine Zeitschätzung kann man getrost verdoppeln oder gleich vervierfachen. Insbesondere, wenn die Anforderungen ändern oder neue dazukommen. Das war aber nicht das einzige Problem.

Obwohl ich C++ nach wie vor für eine sehr mächtige Sprache betrachte, halte ich von Visual C++ und der MFC nicht mehr viel. Eine Entwicklungsumgebung ist so stark wie die Bibliothek, die einem zur Verfügung steht. Und für ein Projekt wie den Resourcewizard erachte ich diese Bibliothek und diese Entwicklungsumgebung als ungeeignet. Das Produkt sollte GUI-gesteuert und möglichst benutzerfreundlich werden; für die GUI-Programmierung muss man unter MFC aber zu oft auf die umständliche Windows-API zurückgreifen. Das Erstellen von speziellen GUI-Logiken ist oft nur „zu Fusse“ zu erreichen und Architektur erscheint mir nicht robust genug für eigene Erweiterungen.

Nichtsdestotrotz konnte ich von diesem Projekt profitieren. Visual C++ scheint trotz seiner Mängel ein Industriestandard zu sein. Meine Erfahrungen mit C++ haben sich bei dieser Arbeit haben sich stark vergrössert - nicht nur wegen den Problemen mit MFC.