Microsoft Foundation Classes 4.0

Aller Anfang ist schwer, besonders der Einstieg in C++, Windows Programmierung und MFC. In diesem Artikel möchte ich anhand eines kleinen Programms zeigen, wie man mit einem modernen Tool wie Visual C++ ein Projekt bearbeitet. Dabei werde ich auf die dahinterliegenden Konzepte eingehen und teilweise auch die entstehenden Beschränkungen beschreiben. Grundlagenwissen über C++ und den Aufbau eines Windowsprogramms erleichtern das Verständnis, auch die Kenntnis des Scribble Tutorials ist hilfreich.

Heinrich Pommer

Die Aufgabenstellung

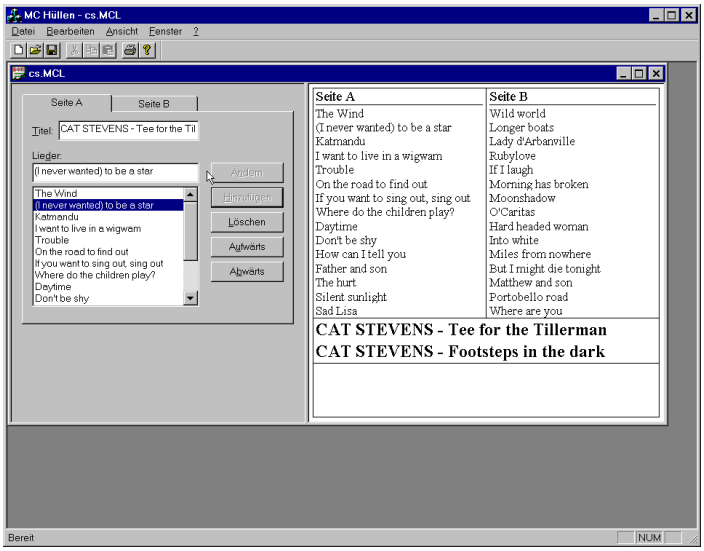

Abbildung 1: **Das Ziel unserer Bemühungen**

Ich möchte ein Programm zum Erstellen von Hüllen für Musikkassetten entwickeln (***Abbildung 1***). Dabei soll es vor allem darum gehen, in einfacher Weise eine Hülle mit Liedtiteln und Kassettentitel zu versehen. Es soll immer dieselbe Schriftart verwendet werden, auch soll keine Rücksicht auf Textformatierungen und Grafiken genommen werden (der Einbau Textverarbeitender Fähigkeiten würde den Rahmen sprengen). Die Dateneingabe soll über ein Formular erfolgen, die fertige Hülle soll jedoch ebenfalls sichtbar sein. Ich weiß, daß es viele und sicherlich bessere Möglichkeiten gibt, dieses Programm zu verwirklichen. Der hier vorgestellte Lösungsvorschlag bietet jedoch den Vorteil, einige Aspekte der MFC-Programmierung zu umfassen.

Der AppWizard

Bei Projektbeginn führt kein Weg an diesem Tool vorbei, erstellt es doch das Grundgerüst einer Anwendung. Unser Projekt soll den Namen "MCLabel" tragen und im Großen und Ganzen die Standardeinstellungen übernehmen. Das heißt, es wird eine MDI (Multiple Document Interface) Applikation in deutscher Sprache, ohne Datenbank- und ohne OLE-Support. Auf Seite 4 des Wizards verbirgt sich unter *Advanced* eine wichtige Dialogbox. Hier werden Dateispezifische Informationen abgefragt. Als Dateierweiterung soll uns "mcl" dienen. Für die Titelzeile wählen wir "MC Hüllen", für *Doc Type Name* (Name eines neuen Dokuments) "MCNeu", für *Filter Name* (Dateityptbeschreibung in der FileOpen Dialogbox) "MC Hülle (*.mcl)", für *File new name* (nur für OLE Server interessant) und *File type name* (wird außerhalb von OLE auch vom Explorer als Typ angezeigt) "MC Hülle". Die Einstellung für *File Type ID* sollte unverändert bleiben, da unter diesem Schlüssel der Dateityp registriert wird. Die Einstellungen dieses Dialogs werden vom AppWizard in einer Stringressource gespeichert. Die Einstellungen auf der folgenden Seite sind Geschmacksache, Kommentare würde ich auf jeden Fall empfehlen. Die MFC in der DLL Variante bietet den Vorteil kleinerer Programme, dem steht eine etwas langsamere Geschwindigkeit und die Notwendigkeit von MFC40.DLL gegenüber. Die letzte Seite des Wizards zeigt die zu erstellenden Klassen und Dateien an. Hier ändern wir die Basisklasse für *CMCLabelView* von *CView* auf *CScrollView.*

Die erstellten Dateien

Werfen wir einen Blick auf die erzeugten Klassen. Den Grundstein legt die in jedem MFC Programm vorhandene Applikationsklasse *CMCLabelApp.* Diese Klasse stellt, wie der Name schon vermuten läßt, das Programm selbst dar. Dies ist auch das einzige globale Objekt.

```
CMCLabelApp theApp;
```

In einem komplizierten Zusammenspiel zwischen Konstruktor und Startup-Code beginnt das Programm zu laufen. Während der Initialisierung wird die Elementfunktion *CMCLabelApp::InitInstance()* aufgerufen. Hier beginnt der für den Programmierer sichtbare Teil der Initialisierung. Die vorhandenen Standardinitialisierung bereitet einen großen Anteil der Funktionalität vor. Der Dokumenttyp wird registriert, *Drag and Drop* wird aktiviert, Kommandozeilenparameter werden ausgewertet und das Hauptfenster der Anwendung wird erzeugt. Dadurch können z.B. Dokumente zum Drucken einfach auf das Druckersymbol gezogen werden. Jeder, der diese Funktionen bereits in C programmiert hat, weiß wieviel Arbeit man sich bereits jetzt erspart hat. Nach Beendigung der Funktion geht die Anwendung in die für Windowsprogramme typischen Nachrichtenschleife über. Die Standardschleife ist in *CWinApp::Run* implementiert, kann aber für Spezialfälle überschrieben werden. In der Funktion *InitInstance* werden auch die wichtigen Dokumenttypvorlagen für die Anwendung generiert:

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
   IDR_MCNEUTYPE,
   RUNTIME_CLASS(CMCLabelDoc),
   RUNTIME_CLASS(CChildFrame), // custom MDI child frame
   RUNTIME_CLASS(CMCLabelView));
AddDocTemplate(pDocTemplate);
```

Diese wenigen Zeilen bilden die Grundlage für das von den MFC zur Verfügung gestellte und sehr leistungsfähige *Document/View* Konzept. Ich komme auf dieses Konzept später noch zurück, zuerst müssen noch einige Dinge über die Klassenhierachie der MFC gesagt werden (***Abbildung 2***).

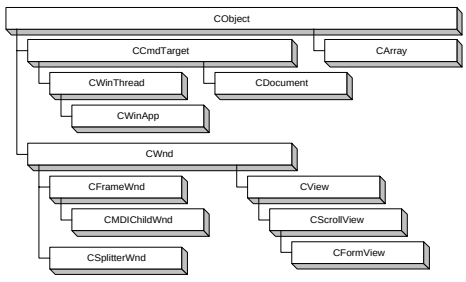

Abbildung 2: **Ein Teil der Klassenhirachie**

CObject: Die Basis allen Seins

Fast alle MFC Klassen sind von der *CObject* Klasse abgeleitet. Diese Vorgangsweise war für die Programmierer der MFC notwendig, um eine *Runtime Type Information (RTTI)* zu realisieren. In die Compilersyntax

wurde dieses Instrument erst in der Version 4.0 von Visual C++ eingebaut. Die RTTI ist notwendig um z.B. Instanzen dynamisch erzeugen zu können. Für jede in der Anwendung vorhandene Klasse, welche von *CObject* abgeleitet ist, wird eine *CRuntimeClass* Struktur erzeugt. Diese Strukturen werden in einer Liste verwaltet. ***Tabelle 1*** zeigt die dazu verwendeten Makros.

Makro	Funktion
DECLARE_DYNAMIC IMPLEMENT_DYNAMIC	erzeugt eine CRuntimeClass Struktur und stellt die Funktion CObject::IsKindOf zur Verfügung.
DECLARE_DYNCREATE IMPLEMENT_DYNCREATE	ermöglicht zusätzlich ein dynamisches kreieren der Objekte
DECLARE_SERIAL IMPLEMENT_SERIAL	erzeugt zusätzlich Code zur Serialisierung
RUNTIME_CLASS	liefert einen Pointer auf die CRuntime Struktur

Tabelle 1: **Hilfsmakros für die Runtime Type Information**

Für das weitere Studium dieser RTTI Implementation empfehle ich [1]. Die *CObject* Klasse hat jedoch noch mehr zu bieten, Diagnosefunktionen und Serialisierung. Auf die Serialisierung komme ich bei der Implementierung der Dokumentklasse noch zu sprechen. Die Diagnosefunktionen dienen zur Überprüfung des internen Status einer Klasse und zur Ausgabe entsprechender Information.

CCmdTarget: Der Nachrichtenempfänger

In dieser Klasse ist das komplette Nachrichtenrouting implementiert. In den MFC können nicht nur Fenster im herkömmlichen Sinn Nachrichten empfangen, sondern auch die Applikations- oder die Dokumentklasse. Sie kümmert sich auch um den richtigen Kontext für die Nachrichten. Ist z.B. ein Menüeintrag nur für ein geöffnetes Dokument sinnvoll, so verbindet man den Nachrichtenbearbeiter mit der Dokumentklasse. Ist nun kein Dokument geöffnet so wird der Menüeintrag automatisch disabled. Auch hier werden intern einige Listen verwaltet. Für tiefer gehenden Informationen ist [2] zu empfehlen. Die endlosen und unübersichtlichen *switch-case* Konstrukte der herkömmlichen C Programmierung gehören damit der Vergangenheit an. Von *CCmdTarget* wird neben der Applikationsklasse, der Dokumentklasse und einigen kleineren Klassen natürlich auch die Fensterklasse *CWnd* abgeleitet.

CWinThread + CWinApp = Programm

Diese beiden Klassen bilden das eigentliche Programm. *CWinApp* enthält unter anderem die Zugriffsfunktionen für die Ressourcen, für die INI Dateien und die *Registry*. Außerdem enthält es die Standardbearbeiter für Datei Öffnen, Datei Neu, Datei Druckereinrichtung und für das Hilfe Menü. *CWinThread* kapselt einen Programmfaden. Will man ein *Multi-Thread* Programm realisieren, so muß jeder Faden von dieser Klasse abgeleitet werden.

CWnd: Die Klassen werden am Bildschirm sichtbar

Diese Klasse kapselt einen Großteil des Windows API's zur Steuerung und Verwaltung der Fenster. Von ihr werden alle Controls, Dialoge, Views und sonstigen Fenster abgeleitet.

Bevor wir uns nun auf das *Document/View* Konzept stürzen werfen wir noch einen Blick auf die vom AppWizard erstellte Klasse *CMainFrame.* Sie stellt das Hauptfenster unserer Applikation dar. Im *OnCreate* Handler werden die Statuszeile und die Toolbar erzeugt. Mehr ist nicht zu tun, wir besitzen eine voll funktionsfähige Toolbar mit Tooltips und eine Statuszeile welche Hilfetext für Menüs und Toolbarbuttons anzeigt. Vor Jahren habe ich einige Stunden damit verbracht, eine Statuszeile in reinem C-Code zu implementieren.

Das Document/View Konzept

Durch die Definition einer Dokumenttypvorlage, wie wir sie in *InitInstance* gesehen haben, werden vier Elemente zu einer Einheit verbunden. Eine Ressourcen ID, eine Dokumentklasse, eine Rahmenfensterklasse und eine Ansichtklasse. Die Ressourcen ID weist dem Dokument ein Menü, eine Accelerator Tabelle (Tastaturabkürzungen), ein Symbol und einen String zu. In dem String stehen die, im AppWizard angegebenen, Informationen über Dateierweiterung, -name usw. Durch diese Verbindungen kann für jedes aktive Dokument der richtigen Kontext im Hauptfenster dargestellt werden.

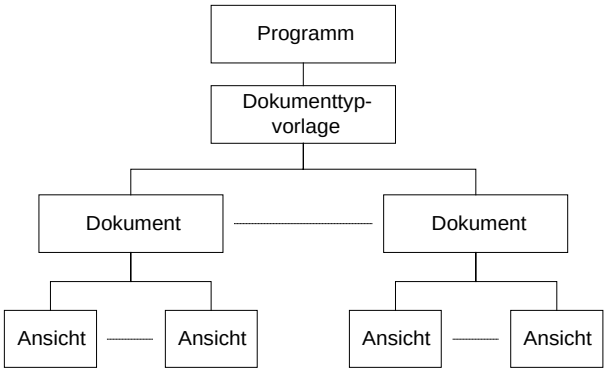

Abbildung 3: **Das Dokument Ansicht Konzept**

Den Namen Dokumentklasse halte ich für schlecht gewählt, besser wäre Dokument Container Klasse. Diese Klasse beherbergt alle zu einem Dokument gehörenden Daten. Sie soll sich um das richtige Speichern und Laden kümmern und die Konsistenz aufrecht erhalten. Die Daten müssen aber nicht, wie in unserem Fall, eine Datei darstellen, sondern können auch wesentlich abstrakter sein, z.B. eine Datenbank, Tabellen und Abfragen daraus oder auch ein Spielbrett bei einem Spiel. Im Dokumentobjekt werden also alle Daten mit denen ein Programm hantiert, abgespeichert. Um die Daten nach Außen zu präsentieren bedient man sich der Ansichtklassen. Sie steuern die Darstellung am Bildschirm und Drucker und legen auch fest, auf welche Weise der Benutzer die Daten verändern kann. Ein Programm kann beliebig viele Ansichten für jedes Dokument zur Verfügung stellen. So stellen z.B. in einem Schachprogramm, das Brett, die Liste der erfolgten Züge und eine Liste der vorgeschlagenen Züge jeweils eine Ansicht dar. Damit die dargestellten Daten über alle Ansichten hinweg konsistent bleiben stellt die MFC spezielle Funktionen zur Verfügung. Die Rahmenfensterklasse dient zu Aufnehmen der Viewklassen und sollte in MDI Anwendungen von *CMDIChildWnd* abgeleitet oder falls keine zusätzliche Funktionalität benötigt wird, diese selbst sein.

CMCLabelDoc: Hier verstecken sich die Daten

Es gibt mehrere Wege die Daten zu definieren. Ich wählte den Weg eines normalen C Arrays für die beiden Seiten, dargestellt durch ein *CArray* vom Typ *CString. CArray* ist eine von den MFC zur Verfügung gestellte Template Klasse für Listen. Deren bietet die MFC im Grunde drei Stück (***Tabelle 2***).

Listklasse	Art der Liste
CArray	Ähnlich eines normalen C Arrays mit Index jedoch mit zusätzlicher Überlaufsprüfung, Einfüge- und Löschfunktion.
CList	Eine doppelt verkettete Liste
CMap	Eine Hash Tabelle

Tabelle 2: **Die Template Klassen der MFC**

Um die MFC Templates verwenden zu können, muß noch die Datei afxtempl.h inkludiert werden. Am Besten in der Datei stdafx.h:

```
#include <afxtempl.h>   // MFC templates
```

Aus früheren Versionen, bevor Templates in den Compiler eingebaut wurden, gibt es auch Listklassen für die Standarddatentypen.

Ich hätte auch für das erste Array die Klasse CArray verwenden können, doch da ich nur zwei Kassettenseiten zu verwalten habe, erschien es mir nicht notwendig. Für die beiden Seitentitel verwende ich wieder ein normales C Array von *CString*. Der nötige Code in mclabeldoc.h sieht folgendermaßen aus:

```
class CMclabelDoc : public CDocument
{
...
// Attributes
public:
    CArray<CString, CString> m_strArray[2];  // Liedtitel
    CString      m_strTitle[2];      // Kassettentitel
...
}
```

Die Deklaration der Variablen kann sowohl von Hand als auch über das Project Workspace Fenster erfolgen. Ich bevorzuge die Methode von Hand, da man nie genau weiß, wo das Developer Studio die Deklaration hinschreibt und die Sourcefiles sollten lesbar bleiben. Am Ende dieser Datei definiere ich auch noch die maximal mögliche Anzahl von Titeln:

```
#define MAX_SONGS    15 // Maximale Anzahl der Liedtitel pro
Seite
```

Die Zahl 15 ergibt sich aus der gewählten Schriftgröße. Für das Dokument muß nun nur noch die Serialisierung eingetragen werden. In die bereits vorhanden Funktion *CMCLabelDoc::Serialize(CArchive& ar)* kommt folgender Code.

```
void CMCLabelDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())        // Richtung des Transfers?
    {
        ar << m_strTitle[0];    // Speichern der Titel
        ar << m_strTitle[1];
    }
    else
    {
        ar >> m_strTitle[0];    // Laden der Titel
        ar >> m_strTitle[1];
    }
    // Speichern bzw. Laden der Listen
    m_strArray[0].Serialize( ar);
    m_strArray[1].Serialize( ar);
}
```

CArchive beschreibt im Normalfall eine Datei, es kann sich aber auch durchaus um einen I/O Stream in einem Netzwerk handeln. Es ist für das Dokument nicht wichtig, wie und wohin es gespeichert wird. „<<“ und „>>“ sind überschriebene Operatoren der Klasse *CArchive* und können für alle Standardtypen verwendet werden. Die Listklassen besitzen bereits eine Serialize Funktion und können sich selbständig speichern und laden. Das Dateiformat ist binär, und unterstützt, sofern die entsprechende MFC Unterstützung aktiviert wurde, auch daß Container Modell von OLE.

Um nun ein Objekt laden zu können, muß es möglich sein, das Objekt dynamisch kreieren zu können, daher die Notwendigkeit der oben erwähnten Makros. Das *IMPLEMENT_SERIAL* Makro bietet auch die Möglichkeit, eine Versionsnummer mitzuführen. Wenn sich ein Projekt im Laufe der Zeit ändert so sollte das Laden alter Dokumente nicht zu einem Programmabsturz führen. Leider funktioniert diese Versionsüberprüfung nur, wenn die Daten in einer eigenen, von *CObject* abgeleiteten, Klasse gekapselt werden. Es ist daher DRINGEND anzuraten, für die gesamten zu serialisierenden Daten eine eigene Klasse zu erstellen. Leider ist dies der MFC als Mißstand anzurechnen, und fördert nicht gerade die Übersichtlichkeit des Projekts. In unserem Programm verzichte ich der Einfachheit halber darauf.

CMCLabelView: Wir wollen auch etwas sehen

Ein Blick auf die vom AppWizard erstellte Datei verrät sofort, hier tut sich wesentlich mehr als bei den anderen Klassen. Neben dem Konstruktor und Destruktor gibt es drei Funktionsrümpfe für die Ausgabe am Drucker. Bemerkenswert ist jedoch die Funktion *OnDraw*. Ihr kommt die wichtigste Arbeit zu, die Ausgabe. Als Parameter erhält die Funktion einen *Device Context Handle*. Hinter diesem Handle kann sich sowohl der Bildschirm als auch der Drucker verbergen. Immer wenn am Bildschirm oder Drucker etwas gezeichnet werden muß, ruft die MFC, nach einiger Vorbereitungsarbeit ihrerseits, diese Funktion auf. Egal ob der Aufruf durch eine Änderung der zugrundeliegenden Daten erfolgt oder durch die WM_PAINT Aufforderung vom Windows System, alles läuft durch die *OnDraw* Funktion. Wir verwenden diese Klasse nur zur Ausgabe, obwohl sie grundsätzlich auch für die Eingabe geeignet wäre. Die Implementierung heben wir uns bis fast zum Schluß auf. Zuerst wollen wir für die Dateneingabe sorgen.

CMCFormView: Ein Dialog im Fenster

Unsere Arbeit für diese noch nicht vorhandene Klasse beginnt mit dem Resourceeditor. Mit dem Dialogeditor entwerfen wir eine Dialogvorlage (***Abbildung 4***).

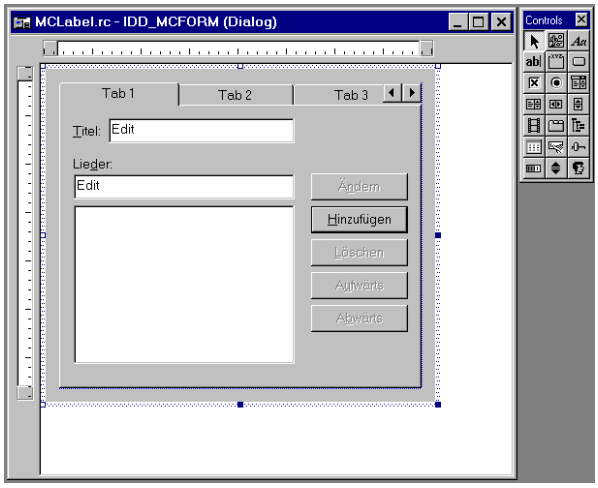

Abbildung 4: **Mithilfe des Dialogeditors erstellen wir unsere Formulareingabe**

Die Standardbuttons für OK und Cancel sollten sofort gelöscht werden. Die Anordnung der Steuerelemente erfolgt einfach mit der Maus. Es ist empfehlenswert, für die ID's der Elemente selbsterklärende Namen zu wählen, um die Übersicht zu waren. Bei noch geöffnetem Dialogeditor rufen wir den *ClassWizard* auf. Dieser erkennt auch sofort den neuen Dialog, und bietet uns die Möglichkeit eine neue Klasse zu erzeugen. Als Namen wählen wir *CMCFormView*, als Basisklasse wählen wir *CFormView*. *CFormView* ist eine von *CView* abgeleitete Klasse mit vollem Support für Formularfenster. Als nächstes wählen wir im *ClassWizard* den Abschnitt *Member Variables.* Wir verbinden nun alle Steuerelemente mit Variablen. Abgesehen von der Listbox verwenden wir überall die Standardkategorie, für die ListBox wählen wir Control als Kategorie. Als Hilfe für die Namengebung soll ***Tabelle 3*** dienen.

Ressource ID	Klasse	Membervariable	Bedeutung
IDC_TABCTL_SELSIDE	CTabCtrl	m_ctlSelSide	für welche Seite gelten die Daten
IDC_BTN_DOWN	CButton	m_ctlDown	Liedtitel nach unten verschieben
IDC_BTN_UP	CButton	m_ctlUp	Liedtitel nach oben verschieben
IDC_BTN_DELETE	CButton	m_ctlDel	Liedtitel löschen
IDC_BTN_CHANGE	CButton	m_ctlChange	Liedtitel ändern
IDC_BTN_ADD	CButton	m_ctlAdd	Liedtitel hinzufügen
IDC_EDIT_SONG	CString	m_strElement	allg. Editelement für Liedtitel
IDC_EDIT_TITLE	CString	m_strTitle	Kassettentitel
IDC_LISTBOX_SONG	CListBox	m_ctlSongList	Liste der Liedtitel

Tabelle 3: **Die Verbindung der Steuerelemente mit Variablen**

Um mit dem Dialog kommunizieren zu können müssen wir noch zwei virtuelle Funktionen überschreiben (*OnUpdate* und *OnInitialUpdate*), wir erledigen dies ebenfalls mit dem *Message Maps* Abschnitt des ClassWizards. Wir müssen auch noch ein paar Nachrichtenbearbeiter implementieren. Alle Buttons brauchen einen Bearbeiter für BN_CLICKED, für die Editelemente benötigen wir EN_CHANGE. Bei der Listbox bzw. den TabControl verwenden wir LBN_SELCHANGE bzw. TCN_SELCHANGE.

Die Implementierung beginnt mit dem Erstellen einer Hilfsfunktion: *GetDocument.* Wir können den Code 1 zu 1 aus den Dateien der *CMCLabelView* Klasse übernehmen, auch das Inkludestatment für "MCLabelDoc.h". Als nächstes benötigen wir eine Hilfsvariable um den Zustand des TabControls verfügbar zu haben, dazu fügen wir folgende Zeilen der Klassendefinition hinzu:

```
protected:
    int m_iSide;
```

Im Konstruktor der Klasse sollte die Variable auf 0 gesetzt werden.

Der Funktion *OnUpdate* kommt zentrale Bedeutung zu. Diese Funktion ist wird von der Dokumentklasse aufgerufen, sobald sich die Daten ändern. Ihr obliegt es, alle Daten des Dokuments in die entsprechenden Variablen bzw. Steuerelemente zu transferieren. Unterstützt wird sie dabei durch die Funktion *UpdateData* welche je nach Aufrufparameter Daten zu den Steuerelementen befördert oder deren aktuellen Wert in den entsprechenden Variablen ablegt. Die MFC verfügen für die Standardelemente über einen DDX/DDV genannten Mechanismus, dieser erlaubt auch eine gleichzeitige Gültigkeitsprüfung der eingegebenen Daten. Wir verwenden diesen Mechanismus nur für die beiden Edit Steuerelemente, alle anderen Steuerelemente sprechen wir direkt über die entsprechenden Klassen an

```
void CMCFormView::OnUpdate(CView* pSender, LPARAM lHint,
CObject* pHint)
{
    CMCLabelDoc* pDoc = GetDocument();
    int i;

    //  Wenn Hilfeinformation vorhanden ist, sich aber
    //  auf die falsche Seite bezieht, gibt's nichts zu tun
    if((lHint & HINT_PRESENT) && ((lHint & 0x1)!=m_iSide))
return;
    //  Ist keine Hilfeinfo vorhanden muß alles geändert werden
    if ( lHint == 0) lHint = HINT_TITLE | HINT_LIST;

    if(lHint & HINT_TITLE) // Titel ändern
        m_strTitle = pDoc->m_strTitle[m_iSide];

    if( lHint & HINT_LIST){ // ListBox füllen
        m_ctlSongList.ResetContent();
        for( i=0; i<pDoc->m_strArray[m_iSide].GetSize(); i++){
            m_ctlSongList.AddString( pDoc-
>m_strArray[m_iSide][i]);
        }
    }
    // Zustände der Button müessen ebenfalls gesetzt werden
    m_ctlAdd.EnableWindow( pDoc-
>m_strArray[m_iSide].GetSize()<MAX_SONGS);
    m_ctlDel.EnableWindow( pDoc-
>m_strArray[m_iSide].GetSize()==0);

    // Übertrage Daten zu den Steuerelementen
    UpdateData( FALSE);
}
```

Um ein ständiges Flackern der Anzeige zu verhindern werden der Funktion *OnUpdate* auch zwei Hilfe Parameter übergeben. Diese sollten von der View Instanz, welche gerade die Daten ändert, gesetzt werden, und von den anderen View Instanzen ausgewertet werden. Wir verwenden hier eine Bitfeld, die Konstanten werden am Sinnvollsten in der Headerdatei der Dokumentklasse definiert.

```
// Bit Null repräsentiert die Seite
#define HINT_PRESENT 0x0002    // Um von Null verschieden zu sein
#define HINT_TITLE   0x0004    // Titel wurde gändert
#define HINT_LIST    0x0008    // Liste der Lieder wurde
geändert
```

Das Tab Steuerelement muß noch für die Verwendung vorbereitet werden, dies geschieht am einfachsten in der Funktion *OnInitialUpdate* welche nur einmal während der Erstellung des Formulars aufgerufen wird:

```
void CMCFormView::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    // Tab Control vorbereiten
    TC_ITEM tcItem = { TCIF_TEXT, 0, 0, "Seite B", 0, 0, 0};
    m_ctlSelSide.InsertItem( 0, &tcItem);
    tcItem.pszText = "Seite A";
    m_ctlSelSide.InsertItem( 0, &tcItem);
    m_ctlSelSide.SetCurSel( 0);
}
```

Die Implementierung der restlichen Funktionen ist in ***Listing 1*** dargestellt und ist selbsterklärend.

Beginn Listing 1>>

```
void CMCFormView::OnBtnAdd() // Liedtitel hinzufuegen
{
    CMCLabelDoc* pDoc = GetDocument();

    UpdateData();   // Daten von den Steuerelementen holen
     // Titel der ListBox hinzufügen
    m_ctlSongList.AddString( m_strElement);
     // Titel dem Array in der CMCLabelDoc Klasse hinzufügen
    pDoc->m_strArray[m_iSide].Add( m_strElement);
    // Status der Buttons updaten
    if( m_ctlSongList.GetCurSel() != LB_ERR)
        m_ctlDel.EnableWindow(TRUE);
    if (pDoc->m_strArray[m_iSide].GetSize()>=MAX_SONGS)
        m_ctlAdd.EnableWindow( FALSE);
    // Editelement für Liedtitel loeschen
    m_strElement = "";
    // Daten wieder zum Steuerelement übertragen
    UpdateData( FALSE);
    // Dokument als verändert markieren
    pDoc->SetModifiedFlag();
    // Die Dokumentklasse muß nun alle anderen Viewklassen informieren
    pDoc->UpdateAllViews(this, HINT_PRESENT | HINT_LIST + m_iSide);
}

void CMCFormView::OnBtnDelete() // Liedtitel loeschen
{
    CMCLabelDoc* pDoc = GetDocument();
    int i = m_ctlSongList.GetCurSel();

    if( i != LB_ERR){
        m_ctlSongList.DeleteString( i);
        pDoc->m_strArray[m_iSide].RemoveAt( i);
        m_ctlAdd.EnableWindow( TRUE);
        m_ctlDel.EnableWindow( FALSE);
        m_ctlChange.EnableWindow( FALSE);
        m_ctlUp.EnableWindow( FALSE);
        m_ctlDown.EnableWindow( FALSE);
        pDoc->SetModifiedFlag();
    }
    pDoc->UpdateAllViews(this, HINT_PRESENT | HINT_LIST + m_iSide);
}

void CMCFormView::OnBtnDown() // Titel in der Liste nach unten bewegen
{
    CMCLabelDoc* pDoc = GetDocument();
    int iPos = m_ctlSongList.GetCurSel();
    CString str;

    if( (iPos == LB_ERR) || (iPos == m_ctlSongList.GetCount()-1))
        return;
    // Weder die ListBox noch die CArray Klasse bieten
    // eine Moeglichkeit Elemente direkt auszutauschen,
    // daher wird ueber eine Zwischenvariable getauscht.
    m_ctlSongList.GetText( iPos+1, str);
    m_ctlSongList.InsertString( iPos, str);
    m_ctlSongList.DeleteString( iPos+2);
    m_ctlSongList.SetCurSel( iPos+1);
    pDoc->m_strArray[m_iSide][iPos+1] = pDoc->m_strArray[m_iSide][iPos];
    pDoc->m_strArray[m_iSide][iPos] = str;
    OnSelchangeListboxSong();
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(this, HINT_PRESENT | HINT_LIST + m_iSide);
}

void CMCFormView::OnBtnUp() // Titel in der Liste nach unten bewegen
{
    CMCLabelDoc* pDoc = GetDocument();
    int iPos = m_ctlSongList.GetCurSel();
    CString str;

    if( (iPos == LB_ERR) || (iPos == 0)) return;
    m_ctlSongList.GetText( iPos, str);
    m_ctlSongList.InsertString( iPos-1, str);
    m_ctlSongList.DeleteString( iPos+1);
    m_ctlSongList.SetCurSel( iPos-1);
    pDoc->m_strArray[m_iSide][iPos] = pDoc->m_strArray[m_iSide][iPos-1];
    pDoc->m_strArray[m_iSide][iPos-1] = str;
    OnSelchangeListboxSong();
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(this, HINT_PRESENT | HINT_LIST + m_iSide);
}

void CMCFormView::OnChangeEditSong() // Editfeld fuer Liedtitel wird
veraendert
{
    int iPos = m_ctlSongList.GetCurSel();
    // Hier wird nur der Buttonstatus geaendert
    m_ctlChange.EnableWindow( iPos != LB_ERR);
}

void CMCFormView::OnChangeEditTitle() // Kassettentitel wird veraendert
{
    CMCLabelDoc* pDoc = GetDocument();

    if (!UpdateData()) return;
    pDoc->m_strTitle[m_iSide] = m_strTitle;
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(this, HINT_PRESENT | HINT_TITLE + m_iSide);
}

void CMCFormView::OnSelchangeListboxSong() // Liedtitel wird selektiert
{
    int iPos = m_ctlSongList.GetCurSel();

    m_ctlSongList.GetText( iPos, m_strElement);
    m_ctlChange.EnableWindow( FALSE);
    m_ctlDel.EnableWindow( TRUE);
    m_ctlUp.EnableWindow( iPos != 0);
    m_ctlDown.EnableWindow( iPos != m_ctlSongList.GetCount()-1);
    UpdateData( FALSE);
}

void CMCFormView::OnSelchangeTabctlSelside(NMHDR* pNMHDR, LRESULT*
pResult)
{   // Auswahl der Kassettenseite wird geändert
    int iSel = m_ctlSelSide.GetCurSel();
```

```
    if( iSel != m_iSide) {
        m_iSide = iSel;
        // Aufruf der eigenen Update Funktion
        OnUpdate( this, 0, NULL);
    }
    *pResult = 0;
}

void CMCFormView::OnBtnChange() // Liedtitel in Liste wird geändert
{
    CMCLabelDoc* pDoc = GetDocument();
    int iPos = m_ctlSongList.GetCurSel();

    UpdateData();
    if( iPos == LB_ERR) return;
    pDoc->m_strArray[m_iSide][iPos] = m_strElement;
    m_ctlSongList.InsertString( iPos, m_strElement);
    m_ctlSongList.DeleteString( iPos+1);
    m_ctlSongList.SetCurSel( iPos);
    pDoc->SetModifiedFlag();
    pDoc->UpdateAllViews(this, HINT_PRESENT | HINT_LIST + m_iSide);
}
```

Listing 1

CSplitterWnd: Wir zerschneiden ein Fenster

Um die Beiden, von uns verwendeten Ansichten gemeinsam in einem Fenster darstellen zu können verwenden wir die MFC Klasse *CSplitterWnd*. Leider ist diese Klasse nicht von *CMDIChildWnd* abgeleitet, so daß wir einen neuen Weg beschreiten müssen, um diese Funktionalität zu nützen. Zu diesem Zweck deklarieren wir eine neue Membervariable von *CChildFrame*:

```
protected:
    CSplitterWnd m_wndSplitter; // Enthält Splitter Window
```

Die Idee dahinter ist, daß wir im Clientbereich unseres MDI Fensters das Splitterfenster generieren und anschließend die beiden Ansichten Fenster als Kindfenster anlegen. Das klingt komplizierter als es in Wirklichkeit ist. Von *CChildFrame* überschreiben wir die virtuelle *OnCreateClient* Funktion.

Die Standard MFC Implementierung dieser Funktion erzeugt einfach das, in der Dokumenttypvorlage, angegebene Ansichtfenster. Dieses Standardverhalten löschen wir und ersetzen es durch folgende Zeilen:

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext)
{
    // Erzeuge Splitter Fenster mit 1 Zeile und 2 Spalten
    // und this als Parent
    if( !m_wndSplitter.CreateStatic( this, 1, 2)){
        TRACE0("CreateStaticSplitter schlug fehl\n");
        return FALSE;
    }

    // In Zeile 0 Spalte 0 erzeuge eine Ansicht von CMCFormView
    if( !m_wndSplitter.CreateView( 0, 0,
    RUNTIME_CLASS(CMCFormView), CSize( 200, 0), pContext)){
        TRACE0("CreateView der 1. Spalte schlug fehl\n");
        return FALSE;
    }
    // In Zeile 0 Spalte 1 erzeuge die Standardansicht CMCLabelView
    if( !m_wndSplitter.CreateView( 0, 1,
        pContext->m_pNewViewClass, CSize( 0, 0), pContext)){
        TRACE0("CreateView der 2. Spalte schlug fehl\n");
        return FALSE;
    }
    return TRUE;
}
```

Nachdem wir noch die Headerdateien der Dokument- und der Formularansichtsklasse der Inkludeliste hinzugefügt haben, wird es Zeit, das Programm zu übersetzen und zu starten. Das Programm besitzt bereits fast die komplette Funktionalität, einzig und allein uns fehlt noch die Kassettenhülle.

Die Zeichenstunde beginnt

Die Implementierung für *CMCLabelView* habe ich in ***Listing 2*** zusammengefaßt. Die GDI Klassen der MFC stellen nur eine hauchdünne Kapselung des Windows API dar, daher erfolgt die Programmierung nach einem ähnlichen Schema. Einzig um das Löschen der angelegten GDI Objekte braucht man sich nicht mehr zu kümmern. Die MFC sorgt dafür, daß alle angeforderten GDI Ressourcen wieder fein säuberlich entfernt werden. In der Funktion *OnInitialUpdate* setzen wir den Mapping Mode auf MM_LOMETRIC, dadurch können die Koordinaten in 0,1 mm Schritten angegeben werden. Ein Standardwerk bezüglich der GDI Programmierung und der Mapping Modes ist [3]. Obwohl es für C und Windows 3.x geschrieben wurde behält es auch unter den MFC seine Gültigkeit. Ich weiß, es ist nicht unbedingt ein schöner Programmierstil, die Koordinaten und Größenangaben direkt in den Code zu schreiben, doch die Werte sollten funktionieren und am einfachsten ist es auch.

Dem aufmerksamen Leser müßte noch die Hint Unterstützung fehlen. Sie wird, wie auch in *CMCFormView,* in der virtuellen Funktion *OnUpdate* ausgewertet. Sie wird immer vor *OnDraw* aufgerufen. Anders als in der Formularansicht werden in dieser Funktion Bereiche des Bildschirms für ungültig erklärt. Es wird zwar im *OnDraw* Bearbeiter keine Rücksicht darauf genommen welche Teile neu zu zeichnen sind, doch es vermindert das lästige Flackern während der Ausgabe erheblich, da Windows alle Zeichenbefehle außerhalb des ungültigen Bereichs ignoriert.

Beginn Listing 2>>

```
void CMCLabelView::OnDraw(CDC* pDC)
{
    CMCLabelDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int i,j;
    // Zeichnen des Rahmens
    CPen penLine;
    if ( !penLine.CreatePen(PS_SOLID, 1, RGB(0,0,0)))
        return;
    CPen* pOldPen = pDC->SelectObject( &penLine);
    pDC->Rectangle( 10, -10,1020,-970);
    pDC->MoveTo( 10, -810); pDC->LineTo( 1020, -810);
    pDC->MoveTo( 10, -680); pDC->LineTo( 1020, -680);
    pDC->MoveTo( 515, -680); pDC->LineTo( 515, -10);
    pDC->MoveTo( 20, -60); pDC->LineTo( 505, -60);
    pDC->MoveTo( 525, -60); pDC->LineTo( 1010, -60);
    pDC->SelectObject( pOldPen);
    // Erstellen eines Font Objekt
    LOGFONT logfnt={ 45, 0, 0, 0, FW_NORMAL, 0, 0, 0, ANSI_CHARSET,
OUT_TT_ONLY_PRECIS, CLIP_TT_ALWAYS, PROOF_QUALITY, VARIABLE_PITCH &
FF_ROMAN, "Times New Roman"};
    CFont fntTop, fntSongs, fntTitle;
    fntTop.CreateFontIndirect( &logfnt);
    CFont *fntOld = pDC->SelectObject( &fntTop);
    // Schreiben der "Seite A, Seite B" Bezeichnung
    pDC->SetBkMode( TRANSPARENT);
    pDC->ExtTextOut( 20, -15, ETO_CLIPPED ,
        CRect( 20, -15, 505, -60), CString("Seite A"), NULL);
    pDC->ExtTextOut( 525, -15, ETO_CLIPPED,
        CRect( 525, -15, 1010, -60), CString("Seite B"), NULL);
    // Erzeugen der Schrift für die Liedtitel
    logfnt.lfHeight = -35;
    fntSongs.CreateFontIndirect( &logfnt);
    pDC->SelectObject( &fntSongs);
    // Schreiben der Liedtitel
    for( j=0; j<2; j++){
        for (i=0; i< min(15,pDoc->m_strArray[j].GetSize()); i++){
            pDC->ExtTextOut( 20+j*505, -65-i*41, ETO_CLIPPED,
                CRect(20+j*505,-65-i*41,505+j*505,-105-i*41),
                pDoc->m_strArray[j][i], NULL);
        }
    }
    // Erzeugen der Schrift für die Kassettentitel
    logfnt.lfHeight = 60;
    logfnt.lfWeight = FW_BOLD;
    fntTitle.CreateFontIndirect( &logfnt);
    pDC->SelectObject( &fntTitle);
    // Schreiben der Kassettentitel
    pDC->ExtTextOut( 20, -685,ETO_CLIPPED,CRect(20,-685,1010,-745),
        pDoc->m_strTitle[0], NULL);
    pDC->ExtTextOut( 20, -750,ETO_CLIPPED,CRect(20,-750,1010,-810),
        pDoc->m_strTitle[1], NULL);

    pDC->SelectObject( fntOld);
}

void CMCLabelView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal;
    // TODO: calculate the total size of this view
    sizeTotal.cx = 1030; sizeTotal.cy = 980;
    SetScrollSizes(MM_LOMETRIC, sizeTotal);
}
void CMCLabelView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    if (lHint != 0){
        // Einstellen des Mappingmodes
        CClientDC dc(this);
        OnPrepareDC( &dc);
        CRect rect;
        // Auswerten der HINTS
        if ((lHint & HINT_TITLE) && !(lHint & 0x1)){
            rect.SetRect( 20, -685, 1010, -745);
            dc.LPtoDP( &rect);
            InvalidateRect( &rect);
        }
        if ((lHint & HINT_TITLE) && (lHint & 0x1)){
            rect.SetRect( 20, -750, 1010, -810);
            dc.LPtoDP( &rect);
            InvalidateRect( &rect);
        }
        if ((lHint & HINT_LIST) && !(lHint & 0x1)){
            rect.SetRect( 20, -65, 505, -679);
            dc.LPtoDP( &rect);
            InvalidateRect( &rect);
        }
        if ((lHint & HINT_LIST) && (lHint & 0x1)){
            rect.SetRect( 525, -65, 1010, -679);
            dc.LPtoDP( &rect);
            InvalidateRect( &rect);
        }
        return;
    }
    Invalidate();
}
```

Listing 2

Was bleibt noch übrig

Die Applikation ist nun fast fertig, zwei kleine Schönheitsfehler gilt es zu beseitigen. Bei beiden Fehlern durchkämmte ich sämtliche Dokumentationen (u.a. die kompletten MS Developers Library mit ca. 500MByte gepackter Information) um auf eine ordentlich Lösung zu stoßen. Vergeblich, für beide Probleme bedurfte es eines Tricks.

1. Problem:

Sobald unser Formular den Eingabefokus besitzt, ist ein Drucken nicht mehr möglich. Klickt man hingegen auf das Ausgabefenster so steht der Druck- und Seitenansichtsbefehl wieder zur Verfügung. Der Grund liegt in der sonst brauchbaren Kontextsensitivität des MFC Nachrichtenverteilers. Die Menü ID's für den Druck sind in *CMCLabelView* mit internen Bearbeitern verbunden. In *CMCFormView* fehlt diese Verbindung. Es nützt auch nichts, die Verbindung herzustellen, da die Formularansicht nicht weiß wie der Druck auszusehen hat. Die logische Folgerung wäre die Verbindung in den Kontext des MDI Fensters oder des Dokuments zu heben. Dieser Versuch schlägt ebenfalls fehl, da der Bearbeiter nur im Kontext einer Ansichtsklasse verwendet werden darf. Das heißt, wir müssen zunächst in der Formularklasse Bearbeiter einbauen. Ein direkter Aufruf der entsprechenden Funktionen in der *CScrollView* Klasse scheitert aufgrund einer *protected* Implementierung. Zur Lösung müssen wir zwei Funktionen in die *CMCLabelView* Klasse einbauen:

```
void CMCLabelView::Print(){    OnFilePrint();}
CMCLabelView::PrintPreview(){  OnFilePrintPreview();}
```

Für die Bearbeiter in *CMCFormView* verwenden wir folgende Zeilen:

```
void CMCFormView::OnFilePrint()
{
    CMCLabelDoc *pDoc = GetDocument();
    POSITION pos = pDoc->GetFirstViewPosition();
    while (pos != NULL)
    {
        CView* pView = pDoc->GetNextView(pos);
        if ( pView->IsKindOf( RUNTIME_CLASS( CMCLabelView))) {
            ((CMCLabelView *)pView)->Print();
            break;
        }
    }
}
```

Zusätzlich müssen wir noch folgende Zeile in den *MESSAGE_MAP* Abschnitt in "*MCFormview.cpp*" einbauen:

```
    ON_COMMAND(ID_FILE_PRINT_DIRECT, OnFilePrint)
```

Da für ID_FILE_PRINT_DIRECT kein Menüeintrag vorhanden ist, kann dieser Berarbeiter nicht über den *ClassWizard* angelegt werden. Für ID_FILE_PRINT und ID_FILE_PRINT_PREVIEW ist der Class Wizard ohne Probleme verwendbar. Die Nachricht ID_FILE_PRINT_DIRECT wird intern abgesandt wenn z.B. ein Druckauftrag per DDE erfolgt (Für das Drucken aus dem Windows Explorer heraus wird DDE verwendet).

Der Bearbeiter für *CMCFormView::OnFilePrintPreview* ist bis auf den Aufruf von *PrintPreview* identisch. Mit den beiden Befehlen *GetFirstViewPosition* und *GetNextView* durchsuchen wir die Liste der aktiven Views nach der gewünschten Klasse und rufen dort unsere Dummyfunktionen auf. Der Nachteil dieser Lösung ist, daß die gewünschte Ansichtklasse geöffnet sein muss (bei unser immer der Fall). Wünschenswert wäre es, wenn es eine Möglichkeit gäbe einen Standardansicht für diesen Fall zu definieren.

2. Problem:

In jedem neuen MDI Fenster erscheint unser Formular verstümmelt. Man kann zwar beim Erzeugen des Formulars (in *CChildFrame::OnCreateClient*) eine Größe angeben, doch gibt es meines Wissens keine Möglichkeit die richtige Größe zu erfahren. Da *CFormView* von *CScrollView* abgeleitet ist scheint es auch, daß es eine passende Funktion gibt: *ResizeParentToFit*. Die Dokumentation schlägt auch die Verwendung dieser Funktion vor, doch trügt der Namen. Die Funktion verändert nicht das Elternfenster sondern das nächst höher liegende *FrameWindow*. In unserem Fall also *CChildFrame*, die Splitterposition bleibt jedoch unverändert. Der Trick wie man die richtige Größe des Formulars ermittelt führt über die Scroll-Leiste. Die Funktion *GetDeviceScrollSizes* liefert uns die maximale Scrollposition und diese entspricht auch der benötigten Fenstergröße. Wir fügen also der Funktion *CMCFormView::OnInitialUpdate* die Zeile:

```
ResizeParentToFit( FALSE);
```

hinzu und der Funktion *CChildFrame::OnCreateClient* die folgende Zeilen:

```
CFormView *pwnd;
CSize siz1,siz2,siz3;
int idummy;
pwnd = (CFormView *)m_wndSplitter.GetPane(0,0);
pwnd->GetDeviceScrollSizes( idummy, siz1, siz2, siz3);
m_wndSplitter.SetColumnInfo( 0, siz1.cx, 0);
m_wndSplitter.SetRowInfo( 0, siz1.cy, 0);
m_wndSplitter.RecalcLayout();
```

Auch diese Lösung ist nicht ganz Wasserdicht. Die Funktion *ResizeParentToFit* kann dazu führen, daß das MDI Fenster größer als das Hauptfenster ist. Dies wirkt sich aber erst bei Bildschirmauflösungen von 800x600 und darunter aus. Das bei diesen Auflösungen der Bildschirm zu klein wird dürfte sowieso hinlänglich bekannt sein.

Möglicherweise gibt es für die zwei Probleme auch einfachere Lösungen und ich sehe vor lauter Bäumen den Wald nicht mehr. Für Anregungen und Hinweise stehe ich gerne per eMail zur Verfügung.

Trotzalledem sei zu sagen, daß die Vorteile der MFC gegenüber der herkömmlichen C Programmierung überwiegen. Man erspart sich wesentliches an Tipparbeit und das Programm bleibt auch übersichtlicher. Daß es natürlich auch Grenzen gibt an die man früher oder später stößt scheint dabei nur logisch.

Literatur

[1] Marcellus Buchheit: Inside MFC Teil 2, System Journal Mai/Juni 1995, S.90

[2] Marcellus Buchheit: Inside MFC Teil 3, System Journal Juli/August 1995, S.116

[3] Charles Petzold: Programing Windows □


```
micado Smalltalker
==================
   MessageBox message: 'Hello, World'


New Manager
===================
   10 PRINT "HELLO WORLD"
   20 END


Middle Manager
===================
   mail -s "Hello, world." bob@b12
   Bob, could you please write
   me a program that prints
   "Hello, world."?
   I need it by tomorrow.
   ^D
```

```
Senior Manager
==================

   % zmail jim
   I need a "Hello, world."
   program by this afternoon.


Chief Executive
===================

   % letter
   letter: Command not found.
   % mail
   To: ^X ^F ^C
   % help mail
   help: Command not found.
   % damn!
   !: Event unrecognized
   % logout
```

```
Assembler Freak
  ===============

   .MODEL SMALL
   .CODE
   org 100h

   mov ah,9
   mov dx,offset hellostr
   int 21h
   mov ax,4C00h
   int 21h

   hellostr db 'Hello World$'

   tasm /mx hello.asm
   tlink hello /t
```