**GUI-Design**

Bei der GUI fiel die Entscheidung auf eine Oberfläche ähnlich der von Visual Studio oder Eclipse, mit einem Editorbereich und Listen zur Ausgabe verschiedener Informationen.

Hierbei bildet die Editor-Komponente den Hauptarbeitsbereich. Diese wird durch Scintilla zur Verfügung gestellt. Alles was hier zu tun ist, ist die Scintilla-Komponente in den Code einzubinden.  
Scintilla hat sogar den Vorteil dass es das Highlighting für verschiedene Sprachen durch so genannte Lexer übernimmt. Es existiert bereits ein MMIXAL-Lexer, das heißt dieser muss nicht extra implementiert werden.

Eine weitere Listbox sorgt für die Ausgabe der bei der Übersetzung entstanden Fehler.  
Diese ist wie die Konsolenausgabe nur sichtbar wenn benötigt. Hier bestand die Überlegung ob die Fehlerliste nicht mit der Konsolenausgabe gekoppelt unter verschiedenen Tabs unterhalb der Editor-Komponente angezeigt wird (ähnlich Visual Studio, NetBeans, Eclipse).

Im Allgemeinen bestand die Überlegung die temporären Komponenten (alles außer dem Editor) unterhalb dem Editor als Tabs anzuzeigen, um so Scintilla möglichst viel Platz zu bieten.  
Da die Textview der eigentliche Arbeitsbereich des Nutzers ist sollte dieser auch der maximal mögliche Platz zugeschrieben werden. Im Endeffekt bedeutet das dass alle anderen Komponenten nur dann angezeigt werden wenn sie tatsächlich benötigt werden.

Das Menü der Anwendung wird relativ einfach gehalten.

Unter dem Menüpunkt „Datei“ wird es die Standardfunktionen „Neu“, „Öffnen“, „Speichern“, „Speichern Als“ und „Beenden“ geben.

Unter „Bearbeiten“ gibt es „Rückgängig“, „Wiederholen“, „Ausschneiden“, „Kopieren“, „Einfügen“, „Löschen“ und „Alles Auswählen“.

Die bisherigen Punkte beinhalten lediglich Funktionen welche mit dem Editor kommunizieren.

Der Menüpunkt „MMIX“ wird die Funktionen „Ausführen“(Run) und „Übersetzen“(Assemble) beinhalten. Diese führen die Befehle aus dem MMIX-Code aus.

Außerdem gibt es noch das Untermenü „Hilfe“ welches die Version- und Hilfsdaten anzeigt.

**Integration der MMIX Funktionalitäten**

Bei der Integration der MMIX-Komponenten boten sich drei Möglichkeiten:

- als Kommandozeilenaufruf im Hintergrund  
 - als interner C Aufruf  
 - als interner Kommandozeilenaufruf

Ersteres wird realisiert als Aufruf, welcher im Hintergrund eine Kommandozeile öffnet, einen Befehl ausführt und den Output in die IDE-Komponente piped. Somit wird die Arbeit an Windows übergeben.

Bei der zweiten Möglichkeit werden die benötigten Teile der MMIX-Umgebung als C-Datei in die IDE eingebunden. Diese werden dann im Hintergrund mit den nötigen Dateien ausgeführt. Hierzu muss der Quelltext der C-Dateien verändert werden sodass der Input nicht von der Konsolenzeile stammt und kein Output auf eben diese zurückgeht.  
Dies wird bewerkstelligt mit Change-Files (.ch). Hierzu später mehr.

Die dritte Variante ist eine Mischung aus den ersten beiden.  
Der Programmcode der MMIX Komponenten ist im Hauptprogramm eingebettet (wie in Möglichkeit 2), d.h. nicht durch außenstehende einsehbar. Bei Aufruf der Funktionalität wird durch die Win32 Anwendung ein Kommandozeilenfenster gestartet. Danach wird die MMIX-Funktion intern gestartet und der Output auf die Kommandozeile gepiped. Zu beachten ist hier: diese Kommandozeile unterscheidet sich von der Kommandozeile in Möglichkeit 1.   
In der ersten Variante wurde Windows der Befehl gegeben eine Funktion auf der Kommandozeile auszuführen. Das Ergebnis wurde danach von dieser abgefragt.   
Hier wird durch das Programm selbst eine Kommandozeile geöffnet aber nicht die Funktion darin ausgeführt sondern nur der Output in das Fenster gepiped. Dabei handelt es sich im Grunde um ein Fenster mit einem speziell stilisierten Textfeld.

Zuerst die Vorteile und Nachteile der drei Möglichkeiten und warum für was entschieden wurde.

**Vorteile des Kommandozeilenaufrufs:**

- leicht einzubauen: Da man hierfür lediglich die exe-Dateien benötigt kann dieser Aufruf sehr leicht eingebaut werden. Im Code wird dies durch lediglich eine Zeile bewerkstelligt. Diese Zeile ruft den Befehl auf und piped gleichzeitig den Output in eine String-Variable.

- keine Änderungen am eigentlichen C-Code: Man muss den C-Code nicht ändern wodurch die exe-Dateien einfach durch den Nutzer ausgetauscht werden können. Sollte eine neue Version der MMIX- Komponenten herauskommen können diese einfach ausgetauscht werden, ohne dass die IDE upgedated werden muss. Wird die DIE   
 dann so intelligent geschrieben dass sie sich automatisch die nächste/vorhandene MMIX-Installation sucht muss der Nutzer der IDE nicht mal sagen wo sich diese befindet.

**Nachteile des Kommandozeilenaufrufs:**

- Im Hintergrund wird ein Kommandozeilenfenster geöffnet: Der Aufruf des Kommandozeilenbefehls verursacht ein tatsächlich „physisches“ Öffnen eines Kommandozeilenfensters. Dieses Fenster trägt nicht gerade positiv zum Erscheinungsbild der Anwendung bei, da man an einigen   
 Merkmalen erkennen kann dass es sich dabei tatsächlich um eine Konsole handelt. Solche Merkmale sind z.B. der Name des Fensters. Im Endeffekt sieht es dann so aus dass sich ein cmd-Fenster öffnet irgendetwas geschieht, aber kein Output angezeigt wird und dann nach kurzer Zeit sich dieses Fenster wieder schließt, worauf dann die eigentliche Ausgabe in der IDE erfolgt.

- Benötigte „Installation“ von MMIX (je nach Bauart der IDE): Wird die IDE so gebaut dass sich die MMIX-Exe-Dateien nicht im selben Verzeichnis befinden müssen, muss entweder der Nutzer dem Programm sagen wo sich die Dateien befinden oder das Programm muss sich diese selbst suchen. Da es die Benutzbarkeit des Programms stark beeinträchtigt und das einfache Bediengefühl darunter leidet, können wir dem Nutzer nicht zumuten der IDE zu sagen wo sich die Dateien befinden. Im Endeffekt muss ein Suchalgorithmus gebaut werden, welcher den ganzen PC durchsucht wenn im vorher gespeicherten Verzeichnis keine MMIX-Dateien gefunden werden konnten.

**Vorteile der internen/integrierten C-Variante:**

- Der Benutzer bekommt nichts vom Aufruf mit: Da die C-Datei intern ausgeführt wird. Es öffnet sich kein Kommandozeilenfenster und es muss auch keine Datei ausgewechselt werden oder der Pfad dafür eingegeben werden.

**Nachteile der internen/integrierten C-Variante:**

- Der C-Code muss zuerst verändert werden: Obwohl sich die Integration des C-Codes auch wieder nur aus einer Zeile besteht muss der Original- MMIX-Code erst einmal umgeschrieben werden sodass er mit den Änderungen klar kommt. Dies geschieht mit sogenannten Change-Files (.ch). Mehr dazu wird später geklärt.

**Vorteile der internen Kommandozeile:**

- Der Benutzer sieht keine MMIX-Programme: Da sich hierfür keine MMIX-  
 Programme im Pfad der Anwendung befinden müssen, sieht der  
 Benutzer dies auch nicht. Es entsteht keine Verwirrung durch zusätzliche Programme im Pfad.

- Visuellen Charme der Konsole bleibt erhalten: Es sieht sehr unsauber aus  
 wenn im Hintergrund der Anwendung eine Konsole geöffnet wird. Dies  
 liegt allerdings nur an der Merkmalen dieser Konsole, wie Name oder History, und der Tatsache dass die Konsole nichts ausgibt. Sind diese nicht vorhanden kann es durchaus seinen Charme haben wenn das Programm eine Konsole öffnet. Dies variiert von Situation zu Situation.

**Nachteile der internen Kommandozeile:**

- Charme der Konsole manchmal fehl am Platz: Wenn es zu manchen Anlässen seinen Reiz hat eine Konsole zu öffnen, gibt es immer noch  
 die anderen Anlässe an denen es unangebracht ist, wenn eine   
 Kommandozeile auftaucht und einem mitteilt was gerade passiert. Wie  
 gesagt ist die Kommandozeile von Situation zu Situation angemessen   
 oder eben nicht.

- Der C-Code muss zuerst verändert werden: Der selbe Nachteil wie bei der   
 internen/integrierten C-Variante.

Im Endeffekt fiel die Entscheidung auf die interne Variante, wenn man vom Assembler redet. Der Simulator (Run) hat die interne Kommandozeile bekommen.  
Auch wenn die externe Variante einfacher einzubauen ist, kann man dem Nutzer den Aufwand nicht zumuten auch noch eine MMIX-Installation zu haben und diese auf dem neuesten Stand zu halten. Der Benutzer geht davon aus dass er eine vollständige Entwicklungsumgebung bekommt, wenn er eine IDE installiert.  
Außerdem gilt es als „unsauber“ wenn ein Programm bei einem Druck auf einen Button eine Kommandozeile öffnet und diese leer wieder schließt ohne zu sagen was vorgefallen war oder was das schwarze Fenster da überhaupt tut.  
Sicher ist es ein massiver Vorteil der externen Variante dass man den eigentlichen C-Code nicht ändern muss. Hierdurch können Fehler durch den Entwickler ausgeschlossen werden.  
Beim Run fiel die Entscheidung auf die interne Konsole da es eben diesen Charme mit sich bringt. Dieser Charme wäre beim Assembler fehl am Platz da das Ergebnis (die Fehler / oder eben keine Fehler) sowieso in der IDE angezeigt werden. Die Konsole wäre beim Assembler schlichtweg nutzlos.

**Änderungen an den MMIX-Funktionen:**

Erst einmal etwas Allgemeines zur Beschaffenheit des Sourcecode von MMIX.

Die Komponenten von MMIX liegen als CWEB-formatierte w-Files vor.  
Die Idee dahinter ist die dass ein w-File sowohl den Sourcecode als auch die Dokumentation dazu beinhaltet.  
Hierbei wird ganz einfach in C programmiert. Zudem können in den Code Verweise auf anderen Code oder dokumentierende Textbereiche eingefügt werden.  
Verweise auf anderen Code bedeutet dass man folgendes schreiben kann:

if( variable == 1 ) {  
 <case One>;  
 } else {  
 <case NotOne>;  
 }

Hier sind <case One> und <case NotOne> Verweise auf existierenden Code. Dieser muss natürlich implementiert werden. Das sieht dann in etwa so aus:

<case One>=  
 return variable++;

<case NotOne>=  
 return 1;

Mit dieser Technik kann man in Schritten programmieren ohne dass man häufig verwendeten Code in Methoden auslagern muss.

Um nun aus dem Ganzen ordentlichen C-Code zu generieren benötigt man ein kleines Programm namens CTangle. Dieses extrahiert den eigentlichen Code aus dem w-File und schreibt es in ein C-File. Während diesem Prozess kann man dem Programm ein so genanntes Change-File mitgeben. Dieses Change-File trägt Informationen welche Teile im Code wie geändert werden sollen.

Bevor mit der Erklärung der Change-Files fortgefahren wird sollte noch angemerkt werden dass es ein Programm namens CWeave gibt welches analog zu CTangle den Dokumentationscode aus dem w-File extrahiert und in ein tex-File schreibt.  
TeX, CWEB und MMIX sind durch Donald Knuth entwickelt worden.

So jetzt aber zu den Change-Files:  
Diese Dateien sind recht einfach aufgebaut. Sie beinhalten eine Liste aus Änderungen. Diese bestehen in sich aus einem Teil der den Original-Text darstellt und einem Teil der den neuen Text repräsentiert.  
Mit „@x“ wird markiert dass hier der Original-Text kommt, „@y“ ist der Marker für den neuen Text und „@z“ signalisiert dass das Ende der Änderung erreicht ist.

Das Ganze sieht dann so aus:

@x  
 alter Text  
 @y  
 neuer Text  
 @z

Hierbei wird der gesamte alte Teil komplett durch den neuen ersetzt. Natürlich müssen die Zeilen exakt sein. Sollte nur ein Leerzeichen fehlen wird der Teil nicht ersetzt und ctangle spuckt Fehler aus.

Multiple Änderungen sind nicht möglich das heißt das automatische Aufspüren einer bestimmten Zeile und ersetzen dieser funktioniert nicht.

Nun zu den Änderungen die im ursprünglichen MMIX gemacht wurden.

**Änderungen in MMIXAL:**

Der Assembler musste so verändert werden sodass er keinen Dateinamen auf der Kommandozeile erwartet sondern gleich die Datei vom Hauptprogramm erhält.  
So wurde die main zu einer normalen Funktion umgeschrieben. Sie erwartet jetzt zwei Parameter:  
Zum einen die Datei. Diese wurde dank Scintilla bereits vorgeladen. MMIXAL muss diese Datei deswegen nicht mehr laden sondern bekommt einfach nur noch den Pointer auf die Datei.  
Der zweite Übergabewert ist der Name der Datei. MMIXAL benötigt den Namen um eine mmo-Datei mit demselben Namen zu erstellen. Es ist wesentlich einfacher den Namen der Datei direkt zu übergeben als diesen durch den Pointer auf die Datei zu erfahren, weswegen über die „Unsauberkeit“ der Redundanz hinweg gesehen wird.

Des Weiteren werden die Fehlermeldungen nicht mehr auf der Kommandozeile ausgegeben sondern per Funktionsaufruf in das Hauptprogramm übermittelt. Diese Funktion wird im Teil „Aufbau der MMIX-Edit IDE“ unter „Zusätzliche Funktionen“ beschrieben.

Da bestimmte Codewörter für Variablen sowohl in der mmixal.w als auch in der mmix-sim.w existieren mussten diese in einer der beiden Dateien deaktivieren.

Alle return oder exit mussten entfernt werden um das Hauptprogramm nicht unerwartet zu beenden.

**Änderungen in MMIX-sim:**

Auch hier musste die Kommandozeilenkommunikation mit direkten Funktionsaufrufen ausgetauscht werden. So bekommt die neue Main nur eine Datei. Diese Datei wurde vorher durch MMIXAL erstellt.

Ebenso wurden wieder alle exit und return entfernt.

Da die Ausgabe nicht in einer Listbox geschieht sondern in einem intern erstelltem Konsolenfenster muss der Output nicht verändert sondern nur vom Hauptprogramm gepiped werden.

**Integration der Scintilla-Komponente:**

Scintilla ist eine Editor Komponente speziell für Source-Editing. Es ist Open Source und wurde von Neil Hodgson entwickelt.

Dank so genannten Lexern kann man schnell zwischen verschieden Programmiersprachen wechseln. Diese Lexer sind ebenfalls in c++ geschrieben. Der MMIXAL-Lexer stammt von Christoph Hösler (Uni Tübingen). Danke an die Seite.

Der Editor kann unter nativen win32 Anwendungen einfach als neues Fenster eingebunden werden. Also mit Hilfe von CreateWindow. Sobald die Scintilla.dll ordentlich eingebunden wurden kann CreateWindow einfach mit L“Scintilla“ aufgerufen werden.

Wie alle win32 Fenster kann man Scintilla Befehle über Messages erteilen.   
Genau so wird Scintilla auch konfiguriert. Wenn Einstellungen zu ändern sind muss eine Message an das Scintilla Fenster geschickt werden.

Im Folgenden werden Nachrichten mit ihrem jeweiligen Zweck genannt. Sind der WPARAM oder der LPARAM nicht genannt werden sie nicht benötigt => 0. Beim WPARAM handelt es sich um den ersten Parameter der LPARAM ist der zweite.

Zu vielen der jetzt genannten Nachrichten mit einem SET im Namen existiert auch eine zugehörige GET Nachricht. Ob es ein GET dazu gibt kann man der Scintilla Doku entnehmen.

Farben werden als RGB definiert. RGB stammt von Windows und kann ganz einfach eingesetzt werden:  
**RGB(0xFF, 0xFF, 0);**erzeugt ein RGB Objekt der Farbe Hellgelb. Wird in einer Nachricht eine Farbe verlangt ist damit ein solches Objekt gemeint.

**Scintilla und Highlighting:**

Um in Scintilla Highlighting zu aktivieren benötigt es mehrere Dinge:

Zum ersten einen Lexer der bestimmt welches Wort oder welcher Codeteil wie formatiert werden soll. Dabei entscheidet der Lexer welche Wörter zusammengehören und einen Kontext bilden. Intern ordnet der Lexer jedem Codeteil (ein „Coderange“ also eine Anzahl an Zeichen) einen Style zu. Die Styles werden vom Lexer definiert.

**SCI\_SETLEXER:** Setzt den Lexer des Editors. Der WPARAM ist der Lexer der genutzt werden soll.

Zum Zweiten benötigt es diese Styles. Zwar definiert der Lexer die Styles aber nicht ihre Beschaffenheit sondern nur den Namen. Der Style muss dann vom Programmierer gefüllt werden. So kann man die Farben, Schriftarten, Schriftgrößen usw. seinen Bedürfnissen anpassen. Man muss diese nicht setzen. Allerdings ist dann der Lexer auch sinnlos eingebunden. Wird ein Lexer eingebunden ohne dass Styles gesetzt sind passiert im Grunde gar nichts. Der Text wird mit der Standard Windows Schrift und in Schwarz angezeigt. Als Randbemerkung der allgemeine Standard der Stylenamen (Naming Convention) ist SCE\_[SPRACHE]\_[TYP] sodass es beim MMIXAL Lexer z.B. folgenden Style gibt: SCE\_MMIXAL\_LABEL.

Mit folgenden Nachrichten kann man Styles ändern (es sind nur die Nachrichten aufgeführt die im Code von MMIX-Edit verwendet werden):

**SCI\_STYLESETFORE:** Setzt die Vordergrundfarbe des Styles.  
**SCI\_STYLESETBACK:** Setzt die Hintergrundfarbe des Styles.  
**SCI\_STYLESETSIZE:** Setzt die Größe des Textes im Style.  
**SCI\_STYLESETFONT:** Setzt die Schriftart des Textes im Style.

Hier noch ein paar weitere interessante Messages zum Thema Styles:

**SCI\_CLEARDOCUMENTSTYLE:** Resetted den Style des Dokuments. Hier ist Folgendes voraus zu sagen: Man setzt den Style eines Dokuments indem man einen Lexer lädt. Im Programm MMIX-Edit wird dieser gesetzt bevor eine Datei geladen wird. Wird nun der Lexer durch einen anderen ersetzt oder der Lexer geladen nachdem eine Datei geladen wurde, muss der Style des Dokuments neu gesetzt werden. Dies passiert mit SCI\_CLEARDOCUMENTSSTYLE.  
Wie oben beschrieben wird hiermit der Style des Dokuments resetted und ein Restyling durch den Lexer angestoßen. In Notepad++ wird diese Nachricht verwendet wenn man das Highlighting des Programmcodes ändert.

**SCI\_STARTSTYLING:** Beginnt das Styling an der angegebenen Position und setzt eine Maske mit der festgelegt wird welche Styles gesetzt werden können.  
Styling basiert auf Styleflags. Jeder Style hat ein Flag. Mit der Maske sagt man was im folgenden Styling an Styles erlaubt ist. Erster Parameter ist die Position und zweiter die Maske.

**SCI\_SETSTYLING:** Setzt einen Style für die folgenden Zeichen. Dabei ist der erste Parameter die Menge an Zeichen (Länge des Strings). Der zweite Parameter ist der Style.

Es gibt noch mehr Style Messages, welche alle auf der Scintilla Doku Seite zu finden sind.

Beibemerkung: Styles werden nicht nur im Textfeld, für den Code, benötigt sondern andere Scintillakomponenten, die dem Editor zusätzliche Funktionen verleihen, verwenden diese ebenfalls um ihren Inhalt zu formatieren. Einige dieser Komponenten werden im späteren Verlauf hier genannt und erklärt.

**Markieren von Zeilen:**

Scintilla bietet sogenannte Marker an. Diese Marker kann man verwenden um im Text wichtige Stellen zu markieren.  
Hierbei stellt Scintilla 32 Marker zur Verfügung. Diese Marker sind nicht vordefiniert. Es handelt sich dabei lediglich um priorisierte Platzhalter. Der Marker mit der Nummer 0 wird als erstes gezeigt, die Nummer 31 als letztes.  
Man kann diese Marker selbst definieren und seinen eigenen Bedürfnissen anpassen. Dies geschieht wieder mit Hilfe der Messages. So kann man am Marker das Symbol oder die Farbe ändern.

**SCI\_MARKERDEFINE:** Setzt einen Marker auf einen bestimmten Markertyp. WPARAM enthält den Markerindex und LPARAM den Markertyp als int.

**SCI\_MARKERSETFORE:** Setzt die Farbe des Vordergrunds eines Markers. Der WPARAM ist wieder der index des Markers. Im LPARAM wird die Farbe als RGB gespeichert.

**SCI\_MARKERSETBACK:** Funktioniert genau wie SCI\_MARKERSETFORE nur auf den Hintergrund.

**SCI\_MARKERSETBACKSELECTED:** Verändert die Farbe des Hintergrunds wenn der Marker ausgewählt wurde. Funktioniert genau wie SCI\_MARKERSETBACK oder SCI\_MARKERSETFORE.

**SCI\_MARKERADD:** Setzt einen bestimmten Marker an eine Zeile des Dokuments. Der erste Parameter ist die Zeilennummer und der zweite die Markernummer.

**SCI\_MARKERDELETEALL:** Löscht Marker im gesamten Dokument. Der WPARAM enthält den Markertyp, welcher gelöscht werden soll. Man beachte: es werden alle Marker des Typs im GESAMTEN Dokument gelöscht. Sollen sämtliche Marker entfernt werden muss -1 als WPARAM mitgegeben werden.

**SCI\_MARKERDELETE:** Entfernt einen Markertyp in einer Zeile des Dokuments. Soll auch hier sämtliche Markertypen entfernt werden muss statt dem Markertyp ein -1 eingefügt werden. Der Markertyp wird im WPARAM gespeichert, während die Zeilennummer im LPARAM liegt.

**Markieren von Text:**

Sehr ähnlich zu den Markern sind die Indikatoren. Scintilla implementiert Indikatoren die, wie die Marker, Stellen im Code markieren. Der Unterschied besteht darin dass ein Marker die ganze Zeile einschließt. Ein Indikator kann an eine bestimmte Position im Code gesetzt werden und hat eine festgelegte Anzahl an Zeichen.  
Somit ist der Indikator besser geeignet um zum Beispiel einzelne Wörter im Code als fehlerhaft zu markieren.  
Man kann aber auch Suchergebnisse mit Hilfe von Indikatoren im Editor markieren lassen.  
Indikatoren sind ähnlich aufgebaut wie Marker. Es gibt 32 separate Indikatortypen die man individuell gestalten kann. Will man einen Indikator im Code setzen muss man ihn dann nur noch mit Hilfe seines Indizes ansprechen.

**SCI\_INDICSETSTYLE:** Setzt den Style eines Indikators. Erster Parameter ist der Index des Indikators und zweiter der Index des Styles. Anmerkung: diese Styles sind nicht die Standard Scintilla Styles sondern vorimplementierte Formatierungen. Darunter sind Styles die den Text z.B. mit einer geraden/gewellten Linie unterstreichen oder eine Box um ihn zeichnen.

**SCI\_INDICSETFORE:** Setzt die Farbe des Indikators. Der erste Parameter enthält den Indikatorindex und der zweite Parameter die Farbe.

**SCI\_INDICSETALPHA:** Setzt den Alphawert (die Transparenz) des Indikators. Parameter: Index des Indikators, Alphawert in int. In dieser Reihenfolge.

In MMIXAL ist es allerdings einfacher Fehler mit Markern zu markieren, da der Assembler nur Zeilennummern ausspuckt. Will man die Fehler mit Indikatoren markieren muss man entweder den Assembler so manipulieren dass er außer der Zeilennummer auch die Position und die Länge des Fehlers mitgibt, was allerdings abhängig vom Fehler schwierig werden kann, oder man parst den Fehlertext und sucht in ihm nach Stichwörtern (dreckige Variante. Es ist immer besser und eindeutiger von der Quelle detailliertere Informationen zu erhalten). Es ist zu überlegen wie (nicht ob) Fehler, in späteren Versionen von MMIX-Edit, mit Hilfe von Indikatoren, markiert werden.

**Copy, Cut, Paste, etc.:**

Die Standard Strg-Kommandos wie Kopieren, Ausschneiden oder Einfügen müssen an die Editor Komponente geschickt werden. Dazu gibt es die Messages **SCI\_COPY**, **SCI\_CUT**, **SCI\_PASTE** und **SCI\_COPYALLOWLINE**.

Zu SCI\_COPYALLOWLINE gibt es zu sagen dass es eine Erweiterung von SCI\_COPY ist. Der einzige Unterschied besteht darin dass es, wenn die Auswahl im Editor leer ist, die gesamte Zeile als Zeile markiert in das Windows-Clipboard kopiert.

**SCI\_UNDO**, **SCI\_REDO**, **SCI\_CANUNDO** und **SCI\_EMPTYUNDOBUFFER** existieren auch. Hier ist der Name beschreibend. SCI\_UNDO ist das Rückgängig-machen-Kommando. SCI\_REDO wiederholt den letzten Befehl. SCI\_CANUNDO fragt nach ob SCI\_UNDO einen Effekt haben würde und SCI\_EMPTYUNDOBUFFER lehrt den Buffer der benutzt wurde um die Befehle rückgängig zu machen. Damit kann hat dann SCI\_UNDO keinen Effekt mehr.

Es gibt mehrere spezielle Copy- und Paste-Methoden. Näheres ist wieder mal in der Scintilla Doku nachzulesen.

**Fehler in Scintilla:**

Läuft ein Scintilla Editor auf einen Fehler trägt er das nicht nach außen sondern speichert den Fehler, wie es bei Win32 Anwendungen üblich ist, intern.

Dieser Fehler kann abgefragt werden:

**SCI\_GETSTATUS:** Fragt den internen Fehlercode ab.  
**SCI\_SETSTATUS:** Setzt den internen Fehlercode.

Hier die Fehlercodes die bis jetzt, in Scintilla, implementiert sind:

**Code 0: SC\_STATUS\_OK:** Keine Probleme.  
**Code 1: SC\_STATUS\_FAILURE:** Ein Fehler (nicht 2) ist aufgetreten.  
**Code 2: SC\_STATUS\_BADALLOC:** Der Speicher ist voll ausgenutzt und kann nicht weiter gefüllt werden.

**White Space Formatierung:**

Zum Formatieren von White Spaces also Leerzeichen und Einschüben existieren Messages wie:

**SCI\_SETVIEWWS:** Bestimmt ob ein White Space angezeigt wird oder nicht. Dabei werden Leerzeichen als kleiner Punkt angezeigt und Einschübe als Pfeil.

**SCI\_SETWHITESPACESIZE:** Setzt die Größe eines Punktes der ein Leerzeichen markiert.

Dies ist besonders bei White Space sensitiven Sprachen sehr nützlich.  
Da MMIX nicht direkt White Space sensitiv ist, sondern nur ein Leerzeichen zwischen den Wörtern benötigt, macht es nicht besonders viel Sinn die Funktion als Standard aktiviert zu haben.  
Die Funktion integriert zu haben kann aber dennoch sehr nützlich sein.

**Verändern des Cursors:**

Der Cursor (Mauszeiger. Nicht zu verwechseln mit dem Caret, dem kleinen Strich der im Editor die Schreibposition anzeigt.) kann mit der  
**SCI\_SETCURSOR** Message verändert werden. Es gibt 8 verschiedene Cursor zur Auswahl. Leider schweigt die Doku zu diesem Thema ein wenig. Deswegen hier die verfügbaren Cursor:  
**-1:** der Standard Texteditor Cursor. Ein Strich der anzeigt dass der Cursor sich über   
 einem Textfeld befindet.

**1:** Der selbe Cursor wie bei -1.  
**2:** Der Standard Mauszeiger.  
**3:** Ein Pfeil nach oben Zeiger.   
**4:** Windows 7/8: Lade-Kreisel. Ältere Windowsversionen: die Sanduhr.  
**5:** Ein horizontal liegender Pfeil mit zwei Enden. Zeigt normalerweise das seitliche   
 Verschieben eines Komponentenrandes an.  
**6:** Der gleiche Pfeil wie bei 5 nur diesmal vertikal stehend.  
**7:** Kein Cursor.  
**8:** Eine Hand mit einem ausgestreckten Zeigefinger.

**Verändern des Caret:**

Den Caret kann man schon ein wenig mehr verändern als den Cursor, da seine Anzeige nicht von Windows übernommen wird sondern von Scintilla selber.

Hierfür gibt es folgende Nachrichten:

**SCI\_SETCARETFORE:** Setzt die Farbe des Caret. Der erste Parameter ist die Farbe.

**SCI\_SETCARETSTYLE:** Setzt die Art des Caret. Dabei wird der Nachricht als erster Parameter ein int mitgegeben welche beschreibt welcher Caret verwendet werden soll. 0 = kein Caret (unsichtbar), 1 = ein Strich (Standard), 2 = ein Block (wie z.B. in DOS oder Konsolen)

**SCI\_SETCARETLINEBACK:** Setzt die Hintergrundfarbe der Zeile in dem sich das Caret gerade befindet. Der erste Parameter ist dabei die Farbe. Anzumerken ist dass diese Änderung nicht sofort sichtbar ist. Man muss dem Editor erst sagen dass er diese Einstellung verwenden soll. Dazu gibt es folgende Message:  
**SCI\_SETCARETLINEVISIBLE:** Teilt dem Editor mit dass die Zeile in der sich das Caret befindet eine andere Hintergrundfarbe haben soll. Der erste Parameter ist ein bool. Ist dieses bool true wird eine andere Hintergrundfarbe verwendet.

**SCI\_SETCARETWIDTH:** Setzt die Breite des strichförmigen Carets. Das funktioniert nur mit dem Strich-Caret. Der Block-Caret ist standardmäßig so breit wie ein Zeichen.

**Selektionen im Text:**

Hier sind ein paar Messages zum Verändern einer Selektion im Text.

**SCI\_SETSELFORE:** Setzt die Schriftfarbe im markierten Teil des Textes.  
**SCI\_SETSELBACK:** Setzt die Hintergrundfarbe des markierten Textes.  
Bei den beiden Messages ist zu beachten dass erst der zweite Parameter die Farbe ist. Der erste Parameter ist ein bool welches angibt ob die Farbe überhaupt verwendet werden soll.

**SCI\_SETSELEOLFILLED:** Soll die Markierung bis an den rechten Rand des Editors gezeichnet werden, wenn die Markierung das letzte Zeichen in der Zeile einschließt, so muss hier nur true als erster Parameter mitgegeben werden.

Ist der Hintergrund im Text farbig hervorgehoben, z.B. durch Highlighting, und will man diesen Effekt erhalten während der Text markiert ist. So kann man folgende Message verwenden:  
**SCI\_SETSELALPHA:** Setzt die Transparenz der Hintergrundfarbe der Markierung.

**Rahmen:**

Scintilla bietet sogenannte Rahmen an. Dabei handelt es sich nicht um Rahmen um den Editor sondern um zusätzliche Spalten links von der Textkomponente des Editors. Hier kann man bis zu vier solcher Rahmen haben. Die Nummern 0, 1 und 2 sind bereits vorbelegt. So dient 0 zur Anzeige von Zeilennummern, 1 ist eine Symbolleiste und 2 zeigt sogenannte folding symbols an, dient also zur Visualisierung von Code-Folding.   
 Bei Code-Folding handelt es sich um die kontextuelle Zusammenfassung von Codeteilen sodass man sie visuell ausblenden kann. Ein Beispiel: man hat drei C, C++ oder Java Funktionen implementiert. Diese können dank Code- Folding von drei Einzelzeilen repräsentiert werden. D.h. man kann diese Funktionen ausblenden indem man neben dem Code auf einen kleinen Knopf drückt wodurch der Editor die Funktionen ausblendet und durch den Funktionskopf ersetzt.  
Scintilla bietet ebenso Code-Folding an. In MMIX macht dies aber eher weniger Sinn. Aus diesem Grunde enthält MMIX-Edit kein Code-Folding. Deswegen wird hier nur erwähnt dass Scintilla dies anbieten würde.   
Wer sich dafür interessiert kann sich darüber in der Scintilla Dokumentation im Bereich „Folding“ schlau machen.

Natürlich kann man den Rahmen auch eigen definierte Zwecke zuweisen. Als Anmerkung: die vordefinierten Rahmen 0 und 2 sind von der Breite auf 0 gesetzt weswegen sie als Standard ausgeblendet sind.

Hier sind einige Nachrichten mit denen man Rahmen bearbeiten kann:

**SCI\_SETMARGINTYPEN:** Setzt den Typ eines Rahmens. Der erste Parameter ist die Rahmennummer (0 - 4) und der zweite der Rahmentyp. Es gibt folgende Rahmentypen:  
 **0: SC\_MARGIN\_SYMBOL:** Ein Symbolrahmen, was dies genau ist sagt die Dokumentation leider nicht.  
 **1: SC\_MARGIN\_NUMBER:** Ein Rahmen mit den Zeilennummern. Diese muss  
 man nicht extra setzen sie werden automatisch durch Scintilla gesetzt.  
 **2: SC\_MARGIN\_BACK:** Ein Symbolrahmen der seine Hintergrundfarbe dem  
 Default-Style anpasst.  
 **3: SC\_MARGIN\_FORE:** Wie 2 nur dass er seine Schriftfarbe dem   
 Default-Style anpasst.  
 **4: SC\_MARGIN\_TEXT:** Ein Rahmen mit durch den Programmierer   
 anpassbarem Text.  
 **5: SC\_MARGIN\_RTEXT:** Wie 4 nur dass der Text rechtsbündig ist.

**SCI\_SETMARGINWIDTHN:** Setzt die Breite des Rahmen.

**SCI\_SETMARGINSENSITIVEN:** Legt fest ob der Rahmen durch die Maus klickbar ist. Die Nachricht die bei einem Klick auf den Rahmen abgefangen werden muss ist SCN\_MARGINCLICK.

**SCI\_SETMARGINCURSORN:** Verändert den Cursor welcher über dem Rahmen angezeigt wird. Normalerweise wird ein umgedrehter Mauszeiger angezeigt.

**SCI\_MARGINSETTEXT:** Schreibt einen mitgegebenen Text in eine angegebene Zeile im Textrahmen. Es wird in alle Textrahmen geschrieben da Scintilla nicht davon ausgeht dass mehrere gesetzt werden.

Der Textrahmen ist gut geeignet um zum Beispiel Fehler des Compilers oder Assemblers an den zugehörigen Codestellen zu zeigen. Der einzige Nachteil besteht darin dass die Rahmen nur links vom Code dargestellt werden können. Dennoch ist es eine sehr praktische Funktionalität, die einem die Implementierung einer Zeilennummernleiste abnimmt.

**Autovervollständigung:**

Scintilla unterstützt Autocompletion. Es bietet eine ListBox an die ausklappt wenn das Programm den Anstoß dazu gibt.

**SCI\_AUTOCSHOW:** Zeigt die Autocompletion Liste an. Diese wird als String im zweiten Parameter mit gegeben. Der Standard Separator der einzelnen Wörter ist das Leerzeichen. Der erste Parameter ist die Anzahl an bereits eingegebenen Zeichen. Als Standard sind die Einträge in der Liste sortiert und die Liste müsste case-sensitiv reagieren.

**SCI\_AUTOCCANCEL:** Blendet die Liste wieder aus.

**SCI\_AUTOCSETSEPARATOR:** Setzt den Separator der Liste die oben bei SCI\_AUTOCSHOW benötigt wird. Standard ist das Leerzeichen.

**SCI\_AUTOCSETCHOOSESINGLE:** Teilt der Liste mit dass wenn sie nur ein Element enthält dieses ohne Anzeigen der Liste geschrieben wird.

**SCI\_AUTOCSETIGNORECASE:** Setzt die Case-Sensitivität. Standard ist true.

**Tooltips im Code:**

Scintilla bietet eine Möglichkeit an Tooltips zu Textteilen anzuzeigen. Ein Beispiel: man gibt einem Schüler eine mms Datei damit dieser sie verändern kann. Er benutzt MMIX-Edit als Editor und möchte jetzt gerne wissen wie GREG im Detail funktioniert. Nun könnte er, mit den Tooltips die Scintilla anbietet, einfach GREG in den Code schreiben es z.B. markieren und eine Taste drücken, worauf dann ein kleines Fenster aufkommt, welches anzeigt was GREG tut und welche Argumente es verlangt.

Um solche so genannten Call Tips zu setzen gibt es mehrere Nachrichten:

**SCI\_CALLTIPSHOW:** Zeigt einen Call Tip an der angegeben Stelle mit einem angegebenen Text an. Hierbei ist der erste Parameter die Position im Text, z.B. durch die Mausposition oder den Caret zu ermitteln, und der zweite Parameter ist ein String mit dem Text, welcher angezeigt werden soll.

**SCI\_CALLTIPCANCEL:** Schließt den Call Tip wieder.

**SCI\_CALLTIPSETHLT:** Setzt einen Bereich im Text des Call Tip der gehighlighted werden soll. Parameter 1 ist die Startposition und Parameter 2 die Endposition, alles relativ im Text des Call Tips.

**SCI\_CALLTIPSETBACK:** Setzt die Hintergrundfarbe.  
**SCI\_CALLTIPSETFORE:** Setzt die Schriftfarbe.  
**SCI\_CALLTIPSETFOREHLT:** Setzt die Schriftfarbe des Highlight Texts.

**Tastaturkommandos und Macros:**

Scintilla Anwendungen können Tastatureingaben des Nutzers simulieren. Dies ist nützlich wenn man z.B. ein Macro Recording System im Programm haben möchte.  
Jede Taste hat hierbei seine eigene Nachricht.

Es gibt nicht nur Nachrichten die Tasten simulieren sondern auch welche die ganze Befehle simulieren. Wie z.B. die Fähigkeit durch Wörter zu springen (in vielen Editoren heutzutage enthalten. Meistens mit Strg + Pfeiltaste ausführbar).

Die jeweiligen Nachrichten (es gibt eine riesige Liste) sind in der Scintilla Dokumentation nachzulesen.

Daraus kann man dann auch Macros erschaffen. Doch das Starten und Stoppen des Macro Recordings bietet Scintilla auch an.

**SCI\_STARTRECORD:** Startet die Aufnahme von Macros.  
**SCI\_STOPRECORD:** Stoppt das Macro Recording.

Dabei wirft Scintilla jedesmal wenn etwas getan wird eine SCN\_MACRORECORD Nachricht. Diese muss vom Entwickler gefangen und verarbeitet werden.

**Drucken des Codes:**

Der Editor hat eine eingebaute Funktion zum Umwandeln des Codes in druckbare Daten.

**SCI\_FORMATRANGE:** Wandelt einen Textbereich in druckbare Pixel um. Der erste Parameter ist ein Flag welches angibt ob gedruckt oder abgemessen werden soll. Der zweite Parameter ist ein spezielles Sci\_RangeToFormat Konstrukt welches Daten wie das Druckinterface und die Anzahl an Zeichen enthällt.

Weitere Nachrichten und Informationen zum Drucken sind in der Scintilla Doku nachzulesen.

**Direkter Funktionszugang:**

Da es manchmal etwas langsam sein kann mit SendMessage zu arbeiten bietet Scintilla eine Möglichkeit an Nachrichten direkt an den Editor zu senden.  
Da die Performance von MMIX-Edit in dem Punkt gut ist benötigt es so ein Konstrukt vorerst nicht. Aus diesem Grund sollte hier nur gesagt sein dass es eine solche Technik gibt und dass diese unter „Direct access“ in der Scintilla Dokumentation nachgelesen werden kann.

**Line Wrapping und zu lange Zeilen:**

Bei Line Wrapping handelt es sich um Möglichkeit eine zu lange Zeile nach einer bestimmten Anzahl an Zeichen abzuschneiden und in der folgenden Zeile fortsetzen zu lassen.

In MMIX kommt dies höchstens bei Kommentaren vor. Wir nehmen jetzt einmal an dass der Nutzer von MMIX-Edit schlau genug ist eine Zeile selber abzuschneiden und auf der nächsten fortzusetzen, wenn diese zu lang sein sollte. Aus diesem Grund enthält MMIX-Edit kein Line Wrapping.   
Es will hier nur erwähnt sein dass Scintilla Line Wrapping anbietet es aber nicht genutzt wird. Näher wird auf dieses Thema in der Scintilla Doku unter „Line wrapping“ eingegangen.

Ebenso bietet Scintilla eine Möglichkeit an zu lange Zeilen dadurch zu kennzeichnen dass entweder ein vertikale Linie im Dokument gezeichnet wird oder die Zeichen die über das Maximum hinausgehen einen farbigen Hintergrund bekommen.

Bei Sprachen wie Java oder C/C++ ist dies sehr nützlich und hilft den Codestyle zu verbessern. Aber wie oben beschrieben kommt das in MMIX so gut wie nicht vor und da angenommen wird dass der Benutzer intelligent genug ist usw. wurde die Funktion nicht eingebaut und wird deswegen hier nur am Rande erwähnt. Wie immer: Wer sich dafür interessiert kann alles unter „Long lines“ in der Scintilla Doku nachlesen.

**Weitere im Code verwendete Nachrichten:**

**SCI\_ADDTEXT:** fügt Text in den Editor ein. Wird beim Laden von Dateien benötigt. Der WPARAM ist die Länge des Textes und der LPARAM enthält den eigentlichen Text.

**SCI\_GOTOLINE:** Setzt den Caret auf die angegebene Zeile. Der WPARAM enthält die Zeilennummer zu der gesprungen werden soll.

**SCI\_SETSAVEPOINT:** Sagt dem Editor dass das Dokument gesichert ist und nicht zwangsgespeichert werden muss. Hat keinerlei Parameter.

**Sonstige interessante Nachrichten:**

**SCI\_GETTEXT:** Gibt Text aus dem Editor zurück. Der erste Parameter ist die Länge des zurückzugebenden Texts. Dabei ist der Text um 1 kürzer und wird von einem 0 terminierenden Zeichen gefolgt. Der zweite Parameter ist ein Pointer auf die Zielstring.

**SCI\_GETLINE:** Ähnlich wie SCI\_GETTEXT holt es Text aus dem Editor. Holt anstatt einer bestimmten Anzahl von Zeichen eine ganze Zeile aus dem Editor. Benötigt nur den Pointer auf den Zielstring.

**SCI\_REPLACESEL:** Ersetzt die aktuelle Markierung mit dem angegebenen Text. Der zweite Parameter ist der Text. Nützlich um ein Find-And-Replace zu implementieren.  
Für ein Replace-All existiert eine perfomantere Lösung die mit so genannten Targets arbeitet. Näheres ist in der Dokumentation von Scintilla nachzulesen

**SCI\_SETREADONLY:** Setzt das read-only Flag des Dokuments.   
**SCI\_GETREADONLY:** Holt das read-only Flag des Dokuments.  
Beide Nachrichten kann man verwenden um Zugriffsschutz auf das geöffnete Dokument zu waren. Man kann es allerdings auch verwenden um eben diesen Schutz zu brechen und gegen andere zu einzusetzen.

**SCI\_INSERTTEXT:** Setzt Text an der angegebenen Stelle ein. Der erste Parameter ist die Stelle. Der zweite Parameter ist einzusetzende Text.   
Mit SCI\_REPLACESEL kann ein Vervollständigung oder Templates realisieren.

**SCI\_CLEARALL:** Diese Funktion lehrt das gesamte Dokument (Sofern das read-only Flag nicht gesetzt wurde). Diese Funktion ist im eigentlichen Sinne nicht hilfreich sondern es ist eher interessant dass sie existiert.

**SCI\_DELETERANGE:** Löscht eine angegebene Menge an Zeichen an der genannten Position.

**SCI\_FINDTEXT:** Sucht einen String im Text. Repositioniert nicht das Caret. Man sollte sich die Scintilla Dokumentation durchlesen bevor man die Funktion verwendet. Der zweite Parameter ist hier ein wenig mehr als nur ein einfacher String.

**SCI\_SEARCHNEXT:** Sucht das nächste Vorkommen des angegebenen Texts.  
**SCI\_SEARCHPREV:** Analog zu SCI\_SEARCHNEXT.  
Wenn eine Textstelle zurückgegeben wurde muss man selbst/der Entwickler das Caret an die Stelle bewegen.

**SCI\_SETOVERTYPE:** Setzt das overtype Flag.  
**SCI\_GETOVERTYPE:** Holt das overtype Flag.  
Das overtype Flag ist die Implementierung des „Ersetzen“. Das heißt ist dieses gesetzt wird beim Schreiben eines Zeichens das Zeichen rechts vom Caret mit dem Neuen ersetzt.

**Aufbau einer einfachen Win32 Anwendung**

Bevor der Aufbau der MMIX-Edit IDE erklärt wird sollte zuerst gezeigt werden wie eine normale minimale Win32 Anwendung aufgebaut ist. In Folgendem wird gezeigt was ein Win32 Programm alles zum „Überleben“ braucht.

**Die Hauptfunktion des Programms:**

Jedes Programm hat einen Eintrittspunkt in die Anwendung. Bei Java ist es die   
 **int main([…])**  
in Win32 Anwendungen ist es   
 **int WINAPI WinMain([…])**

Im Gegensatz zu Java hat diese Funktion andere Parameter.

Diese sind:

**HINSTANCE:** Das ist die derzeitige Instanz des Programms

**HINSTANCE:** Die vorherige Instanz des Programms (NULL)

**LPSTR:** Die Kommandozeile mit der das Programm aufgerufen wurde

**Int:** Ein Parameter der angibt wie die Anwendung gestartet werden soll.   
 Dadurch kann man zum Beispiel das Programm starten ohne dass  
 es den Fokus erhält.

Für die erweiterten Windows Funktionen benötigt es den include der Windows.h.

**Die Hauptstruktur des Fensters:**

Um jetzt ein Fenster zu starten muss in der WinMain eine WNDCLASSEX erstellt werden.   
Mit  
 **WNDCLASSEX windowClass;**erstellt man eine Struktur die wichtige Informationen des Fensters enthält.

Diese Struktur hat Attribute welche die Informationen speichern. Diese Attribute sind public sodass man sie einfach durch  
 **windowClass.attributeName = attributeValue**verändern kann.

Folgende Attribute müssen auf jeden Fall geändert werden:

**windowClass.cbSize:** Setzt die Größe der Struktur.  
 Sollte auf **sizeof(WNDCLASSEX)** gesetzt werden.

**windowClass.lpfnWndProc:** Hier muss die WndProc Funktion übergeben  
 werden. Ohne diese Funktion reagiert das Fenster zum Beispiel nicht  
 auf Größenänderungen oder auf das Drücken auf Buttons.

**windowClass.hInstance:** Dient zum Übergeben der Instanz des Fensters.   
 Diese Instanz bekommt man von der WinMain Funktion. Dadurch dass  
 man windowClass in der WinMain Funktion erstellt hat kann man der  
 Struktur bei hInstance gleich den ersten Parameter der WinMain  
 übergeben.

**windowClass.lpszClassName:** Das ist der interne Name des Fensters. Nicht   
 zu verwechseln mit dem Namen der auf dem Fenster also auf der GUI  
 steht.

Um das Standard-Aussehen der Anwendung zu ändern gibt es folgende Attribute:

**windowClass.style:** Mit Hilfe des Style Attributs kann man die grundsätzliche  
 Darstellung des Fensters regeln. So kann man ihm einen eingerückten  
 Rahmen oder gar keinen geben. Das Style Attribut kann aber noch viel  
 mehr. Sämtliche möglichen Änderungen sind in der MSDN  
 nachzulesen.

**windowClass.hIcon:** Setzt das Icon der Anwendung. Wenn hier NULL  
 übergeben wird gibt Windows dem Programm ein Standard Icon.

**windowClass.hIconSm:** Setzt das kleinere Icon der Anwendung.   
 Analog zu hIcon.

**windowClass.hCursor:** Setzt den Cursor der Anwendung. Dieser Cursor ist nur sichtbar wenn sich der Mauszeiger im Programmfeld befindet.

**windowClass.hbrBackground:** Setzt den Hintergrund der Anwendung.   
 Windows bietet hier Standard Brushes. Bei der Verwendung dieser ist   
 aber zu beachten dass 1 zu dem Wert hinzu addiert werden muss.

Die Struktur hat noch ein paar Attribute mehr. Wenn man in Visual Studio eine neue Win32 Anwendung erstellt wird die Struktur automatisch mit erstellt und befüllt.  
Näheres ist nachzulesen in der MSDN.

**Das Registrieren von Fenstern:**

Die WNDCLASSEX Struktur muss jetzt auch noch registriert werden.  
Dies tut man mit Hilfe der  
 **Atom WINAPI RegisterClassEx([…])**Funktion. Der Parameter ist:

**const WNDCLASSEX \*:** Ein Pointer zu der eben erstellten Fenster Struktur.  
 Die Struktur sollte gefüllt sein bevor sie registriert wird. Um den Pointer  
 der Struktur zu übergeben kann man **&windowClass** verwenden.

Die Funktion gibt ein ATOM zurück welches Später in zum Beispiel der CreateWindowEx Funktion verwendet werden kann. Dieses ATOM muss aber nicht aufgehoben werden.

Das Einzige das abgefangen werden muss ist ob das Registrieren erfolgreich war oder nicht.

War das Registrieren nicht erfolgreich gibt die Funktion NULL zurück. Mit Hilfe der GetLastError Funktion kann dann der Fehler abgefragt werden. Näheres zu der Abfrage und Behandlung dieser Fehler im Abschnitt „Fehlerbehandlung“.

**Die Erstellung von Fenstern:**

Nachdem die Klasse jetzt registriert ist kann das Fenster erstellt werden.  
Dies passiert mit:  
 **HWND WINAPI CreateWindow([…])**Diese Funktion hat insgesamt 11 Parameter. Die zwei wichtigsten sind:

**1te Stelle: LPCWSTR:** Dient zur Übergabe des Klassennamens des Fensters.  
 Dabei handelt es sich um den zu ladenden Typ der Komponente. So  
 kann man z.B. eine Listbox mit L“ListBox“ laden lassen.

**3te Stelle: DWORD:** Hier wird der Style des Fensters angeben. Dabei handelt   
 es sich im Gegensatz zu vorhin nicht um die Darstellung des Fensters  
 intern sondern um den Hauptrahmen den Windows außen herum packt.  
 So kann man das Fenster als Kind oder als Fenster mit kleinem  
 Rahmen erstellen.

**8te Stelle: HWND:** Das Superfenster. Diesem Fenster wird das neue Fenster  
 als Kind zugeordnet. Das ist nur wichtig wenn man bei den Styles den  
 Wert WS\_CHILD angegeben hat.

**10te Stelle: HINSTANCE:** Der Parameter dient zur Übergabe der  
 HINSTANCE aus der WinMain.

Die Funktion verhält sich ähnlich wie RegisterClassEx. Das heißt dass sie im Erfolgsfall ein HWND zurückgibt und im Falle des Misserfolgs NULL.

Das HWND ist ein Handle auf das Fenster. Dieses Handle sollte zwischengespeichert werden, da man damit später auf das Fenster zugreifen kann.

Wird NULL zurückgegeben kann man mit Hilfe von GetLastError wieder abfragen was für ein Fehler aufgetreten ist. Näheres zu GetLastError und der Fehlerbehandlung unter dem Abschnitt Fehlerbehandlung.

Weiteres zur CreateWindow und CreateWindowEx Funktion kann ebenfalls in der MSDN nachgeschlagen werden.

**Win32 Ressourcen:**

In Win32 Anwendungen gibt es zwei Möglichkeiten Komponenten, Fenster und Menüs zusammen zu verbinden.

Die direkte Möglichkeit beinhaltet das Erstellen der Komponenten, Fenster und Menüs (im Folgenden nur noch Elemente) durch CreateWindow und äquivalente Methoden.  
Dabei muss man in einem riesigen Methodenaufruf angeben wie das Element geformt sein muss und welche Eigenschaften vorhanden sein müssen.  
Um das Element dann anderen Elementen unterzuordnen oder zu positionieren muss man wieder andere Methoden aufrufen.

Eine übersichtlichere und einfach anzuwendende Art ist so genannte Ressourcen zu erschaffen. Diese Ressourcen sind nichts anderes als xml Dateien welche die Elemente im Detail beschreiben.  
Was man vorher alles per Hand in einen Funktionsaufruf schreiben musste kann man hier in übersichtlicher (und besser wartbarer) Form zusammenschreiben.  
Hat man eine IDE wie Visual Studio muss man diese Dateien nicht einmal schreiben. Visual Studio besitzt einen Ressourceneditor. Dieser bietet ein visuelles Interface mit dem man die Ressourcen, per Drag-n-Drop, zusammensetzen kann.

Beide Varianten haben natürlich wieder ihre Pros und Contras:

**Vorteile der Variante mit Methoden:**

- Es ist schneller geschrieben: Es ist schneller eine ListBox mit Hilfe von  
 CreateWindow zu erstellen und später anderen Komponenten   
 unterzuordnen und zu repositionieren.

- Volle Anpassbarkeit: Mit CreateWindow erstellte Elemente sind einfacher   
 seinen Bedürfnissen anpassbar. Das liegt daran dass das Fenster  
 Handle offen liegt und nicht erst beschafft werden muss.

**Nachteile der Variante mit Methoden:**

- Unübersichtlich: Der CreateWindow Methodenaufruf ist extrem  
 unübersichtlich. Dank der Fülle an Parametern kann man schnell  
 mal den Überblick verlieren. Außerdem wirkt der Code sehr zerrissen  
 wenn sich eine 5-10 Zeilen lange Methode in ihm befindet (Sofern man versucht die 80-Zeichen-pro-Zeile Regel einzuhalten).

**Vorteile der Ressourcen:**

- Einfach zu erstellen: man muss sich nicht erst stundenlang durch die  
 Windows Dokumentation lesen bevor man ein halbwegs akzeptables  
 Fenster zustande bringt. Besonders der Ressourceneditor von Visual  
 Studio ist sehr nützlich und mit wenigen Handgriffen hat man sich das  
 passende Element zusammengestellt.

- Austauschbar: Hat man in einem Programm ein tolles Element erstellt, das  
 man später gerne in einem anderen Programm verwenden möchte,  
 muss man nur die rc Datei (Die Ressourcen Datei) kopieren. Nun kann  
 man in dem anderen Programm die selben Ressourcen mit den selben  
 IDs ansprechen.

**Nachteile der Ressourcen:**

- Relativ langsam und komplex für einfache Elemente: Will man nur ein  
 einfaches Element erstellen, also ein Element ohne jegliche Kind Elemente, sind die Ressourcen sehr langsam zum Handhaben.  
 Man hat mit CreateWindow schneller eine ListBox erstellt und dem   
 Hauptfenster als Kind zugeordnet als man dies mit einer Ressource   
 könnte.  
 In dem Sinne ist die Ressource eher für komplexere Bäume von Elementen gedacht.

- Eingeschränkte Anpassbarkeit: Da man von Anfang an kein Fensterhandle  
 hat kann man nicht direkt mit dem einzelnen Element arbeiten. Manche  
 Methoden bieten zwar an ihnen, statt einem Handle auf das Fenster/Element, eine Ressourcen-ID mitzugeben aber dennoch gibt es  
 Methoden welche dies nicht tun. So muss man sich zuerst das Handle  
 auf das Element beschaffen.  
 Dies ist nicht der Sinn der Sache gewesen. Der Sinn hinter den  
 Ressourcen war eine Möglichkeit zu haben mit der man einfach und   
 exakt ein Element erschaffen kann und danach nicht mehr berühren muss. In der modernen Militärtechnik nennt man so etwas   
 Fire-And-Forget.  
 Man erschafft etwas und muss sich danach nicht mehr darum   
 kümmern.

Abschließend ist noch anzumerken dass jedes Element in einer Ressource eine ID haben kann. Diese kann durch den Entwickler gesetzt werden. Jedes Element ist mit dieser ID direkt ansprechbar. Dabei hat man allerdings nicht die Freiheit wie es ein Handle auf ein Fenster bietet. Wie gesagt war das auch nicht die Intention, dass man mehrere Handles auf verschiedene Fenster mit sich durch das Programm zieht.

**Das Erstellen von Dialogen aus Ressourcen:**

Hat man innere Fenster, die durch Ressourcen beschrieben werden, können diese durch   
 **HWND WINAPI CreateDialog([…])**erstellt werden. Hier die Parameter:

**HINSTANCE:** Ein Handle zur Instanz des Programms. Hier ist wieder die  
 Instanz aus dem WinMain zu verwenden.

**LPCTSTR:** Hier ist mit Hilfe von **MAKEINTRESOURCE** die Ressource zu laden die dann an die CreateDialog Funktion übergeben wird.

**HWND:** Eine Repräsentation des Über-Fensters. Das neue erstellte Fenster  
 wird dann als Kind dieses Fensters eingetragen.

**DLGPROC:** Hier kann man die Hauptprozedur des neuen Dialogs eingeben.  
 Diese Hauptprozedur ist beim Dialog das Äquivalent zur WinMain.  
 Es muss keine Prozedur mitgegeben werden, stattdessen kann man   
 auch NULL übergeben.

Bei CreateDialog gilt das Gleiche wie bei CreateWindow: Die Funktion gibt ein Handle auf das neue Fenster (den neuen Dialog) zurück. Dieses Handle sollte gespeichert werden um später das Fenster ändern zu können.

**Menüs aus Ressourcen:**

Um Menüs aus Ressourcen zu erstellen muss man nur ein Menü z.B. im Ressourceeditor erstellen und dieses dann in der WNDCLASS Struktur als Menü mitgeben.

Das interessante daran ist nicht das Erstellen sondern das Ansprechen der einzelnen Funktionen und welche Nachrichten gefangen werden müssen.  
Erstellt man ein Menü mit Hilfe einer Ressource muss man jedem Menüeintrag eine ID mitgeben. Muss ist hier sehr weit gegriffen. Das Programm/Der Editor zwingt einen nicht dazu und die Ressourcen Datei ist nicht fehlerhaft wenn man keine ID mitgibt. Dennoch wäre es etwas contra produktiv wenn keine ID eingetragen wurde, da diese in der Nachrichtenschleife als Wert mitgegeben wird.  
Demnach muss in der Nachrichtenschleife die ID des jeweiligen Eintrags gefangen werden wenn man die zugehörige Funktion implementieren will.

**Accelerator und Tastenkombinationen:**

Hat man ein Menü erstellt möchte man den einzelnen Menüelementen sicher auch Tastenkombinationen mitgeben sodass diese ohne einen Mausklick aktiviert werden können. So kann man Strg+S drücken um ein Dokument zu speichern oder Strg+C um einen Text zu kopieren.

Dies alles erreicht man indem man so genannte Accelerator erstellt.  
Wenn man mit Ressources arbeitet ist es eigentlich recht einfach. Hat man ein Menü mit Visual Studio erstellt, hilft einem die IDE einem indem sie automatisch aus dem Menü bereits eine Accelerator-Ressource erstellt. Hat man keine Ressource die Accelerator beschreibt kann man diese einfach durch Visual Studio erzeugen lassen und dann per Hand füllen. Diese Ressource muss dann nur noch mit  
**HACCEL WINAPI LoadAccelerators([…])**eingelesen werden.   
Diese Funktion nimmt die Instanz des Programms an und eine durch MAKEINTRESOURCE formatierte ID der Ressource.  
Der Aufbau der Einträge kann in Visual Studio nachgesehen werden. Der Accelerator-Editor ist dabei eine sehr nützliche Hilfe.

Arbeitet man nicht mit Ressourcen muss man mit  
**HACCEL WINAPI CreateAcceleratorTable([…])**  
eine Accelerator Tabelle erstellen und diese mit ACCEL Objekten füllen.  
Die Parameter von CreateAcceleratorTable sind ein Array aus ACCEL Objekten und die Anzahl an Objekten die sich im Array befinden.  
Ein solches ACCEL Objekt benötigt bei der Instanziierung drei Parameter. Der erste sind Flags welche angeben wie die Taste gedrückt wird. Dazu gehört auch ob z.B. die Strg-Taste gehalten werden muss während die Taste gedrückt wird. Zum zweiten benötigt es die Taste selber. Hierbei sind die Tasten mit einem Code ansprechbar. Die meisten Tasten wie Zahlen oder Buchstaben benötigen nur das jeweilige Literal. Bei tasten wie der Return Taste oder den F-Keys bedarf es eines speziellen Codes. Diese sind in der Form VK\_[KEY], so hat die Return Taste den Code VK\_RETURN.  
Der dritte Parameter ist die ID des Menüeintrags, also was der Menüeintrag oder die Tastenkombination machen soll. Diese ID wird in der Nachrichtenschleife im Low-Word abgefangen.

Die zweite Variante ist außerdem die einzige Möglichkeit Acceleratoren zu erstellen welche dynamisch während der Laufzeit geändert werden können.  
Wenn man z.B. einem Nutzer die Möglichkeit bieten will, Tastenkombinationen für bestimmte Funktionen des Programms setzen zu lassen.

**Das Positionieren und Skalieren von Fenstern:**

Da man mit CreateDialog nicht wie bei CreateWindow die Größe und Position des Fensters bestimmen kann, muss man das erstellte Fenster später in der Größe nachrichten.

Dies passiert mit:  
 **BOOL WINAPI SetWindowPos([…])**Die Funktion erwartet das Fenster Handle, ein Handle auf ein Fenster dem das erste Fenster vorangestellt werden soll oder ein spezieller Parameter, die Position mit x und y, die Größe in Breite und Höhe und Flags die das Positionieren und Skalieren beeinflussen.

Der letzte Parameter kann 0 gesetzt werden wenn man einfach nur Positionieren und Skalieren will.

All das ist natürlich nachzulesen auf der MSDN Seite.

**Das Anzeigen und Verbergen von Fenstern:**

Nachdem die Fenster jetzt registriert und erstellt sind kann man sie anzeigen. Windows bietet einem da folgende Methode:  
 **BOOL WINAPI ShowWindow([…])**Dabei zeigt die Funktion das Fenster nicht einfach an sondern ändert nur den Sichtbarkeits-Zustand des Fensters.

Wie das Fenster danach sichtbar ist wird mit Hilfe der Parameter geregelt:

**HWND:** Hier wird das Fenster Handle mitgegeben. Deswegen war es beim  
 Erstellen wichtig dass das Handle aufgehoben wurde.

**int:** Das ist der Parameter welcher angibt wie das Fenster gezeigt wird.   
 Möglich ist es hierbei das Fenster maximiert anzuzeigen oder zu  
 verstecken/minimieren. Der Wert welcher zum normalen Anzeigen  
 benötigt wird ist allerdings **SW\_SHOW**.

**Die Nachrichtenschleife:**

Als letztes in der WinMain benötigt es eine Nachrichtenschleife. In dieser wird nichts anderes getan als nach Nachrichten zu fragen und diese zu verarbeiten.

So besteht die Schleife tatsächlich aus nichts anderem als einem GetMessage in einem while Header und in der Schleife das Weiterleiten der abgefangenen Nachricht an die verantwortlichen Stellen.

**Das Erhalten von Nachrichten:**

Das GetMessage ist folgendermaßen aufgebaut:  
 **BOOL WINAPI GetMessage([…])**Die Parameter sind:

**LPMSG:** Ein Pointer auf eine Variable des Typs MSG. In diese Variable wird  
 durch die Funktion die Nachricht gespeichert.

**HWND:** Handle zu einem Fenster von dem die Nachrichten abgefangen   
 werden sollen. Sollen alle Nachrichten aller Fenster des Threads   
 abgefangen werden kann man hier NULL mitgeben.

**UINT:** Der minimale Wert der Nachricht. Dient zum Filtern der Nachrichten.

**UINT:** Der maximale Wert der Nachricht. Ebenfalls ein Filterwert.

Die Funktion holt sich aus der Messagequeue die oberste Nachricht und speichert sie in die durch den ersten Parameter angegeben Variable. Dabei wird nur die Nachricht abgefragt die zu dem jeweiligen Handle passt und zwischen den Filterwerten liegt.

Zu den Filterwerten:

Jede Nachricht hat einen Zahlenwert. Mit Hilfe der beiden Filterwerte kann man angeben in welchem Zahlenbereich sich die Nachrichten befinden müssen.

So kann man mit dieser Funktion bestimmte Nachrichtengruppen oder sogar nur eine Art von Nachricht akzeptieren.

**Das Weiterleiten von Nachrichten:**

Damit die Nachrichten in der Schleife an die richtige Stelle, also an die WndProc, geschickt wird, muss in der Nachrichtenschleife folgende Funktion aufgerufen werden:  
 **LRESULT WINAPI DispatchMessage([…])**Diese Funktion macht nichts anderes als die Nachricht an die richtige WndProc Funktion weiter zu leiten.

Der Parameter ist:

**const MSG \*:** Die Nachricht die weiter geleitet werden soll.

So muss dann einfach nur DispatchMessage(&msg) in der Nachrichtenschleife aufgerufen werden damit die Nachrichten an die WndProc geschickt werden.

**Die Verarbeitung von Nachrichten:**

Diese WndProc ist eine spezielle Funktion zur Verarbeitung der Nachrichten. Diese Funktion muss genau wie die WinMain selbst geschrieben werden. Die Funktion muss folgenden Header haben:  
 **LRESULT CALLBACK WndProc([…])**

Die Parameter hier sind:

**HWND:** Das ist ein Handle zum geöffneten Fenster. Also zu dem  
 Fenster welches die WndProc enthällt.

**UINT:** Die Nummer der Nachricht die geschickt wurde.

**WPARAM:** Der WPARAM mit dem zusätzliche Informationen zur Nachricht  
 mitgegeben werden.

**LPARAM:** Der LPARAM mit dem zusätzliche Informationen zur Nachricht  
 mitgegeben werden.

Der WPARAM und LPARAM werden von verschiedenen Nachrichtentypen  
verwendet um zusätzliche Informationen, wie zum Beispiel welche niedere Nachricht aufgerufen wurde oder welches Element im Fenster die Nachricht ausgelöst hat, mit zu geben.

Die Nachrichtennummern können auf der MSDN Seite nachgesehen werden.

Zu beachten ist dass z.B. Nachrichten von Unterkomponenten des Programms in einer Nachricht verpackt sein können.

Der WPARAM und der LPARAM können in sich selbst unterteilt sein. So kann man in jedem der Parameter zwei Werte verstecken. An diese Werte kommt man mit den Macros **HIWORD** und **LOWORD**. HIWORD holt die erste Hälfte des Parameters und LOWORD dann halt die zweite Hälfte.

**Fehlerbehandlung:**

Die meisten Fehler die abzufangen sind, sind Fehler beim Erstellen von Fenstern.   
Wurde ein Fenster nicht erstellt will man natürlich wissen warum dies so ist.  
Dabei muss man zuerst den Fehler abfragen und ihn dann in eine durch Menschen lesbare Form umwandeln.  
Dies funktioniert natürlich auch bei Fehlern die nicht bei der Fenstererstellung entstanden sind.

Zuerst benötigt man eine Variable zum Speichern des lesbaren Strings:  
 **LPTSTR text**Das wird später für die FormatMessage Methode benötigt.

Zur Abfrage des Fehlers existiert folgende Methode:  
 **DWORD WINAPI GetLastError()**Diese Methode holt den letzten Fehler des Threads und gibt seine Kennnummer als DWORD zurück.  
Die Methode kommt ohne Parameter aus da hier nur der letzte Fehler abgefragt wird.

Da hier nur eine Kennnummer für den Fehler zurückgegeben wird kann der Nutzer damit nichts anfangen. Um eine durch Menschen lesbare Form zu erhalten wird eine weitere Funktion benötigt:  
 **DWORD WINAPI FormatMessage([…])**Die wichtigsten Parameter sind:

**1te Stelle: DWORD:** Flags die angeben wie der Fehler zu verarbeiten ist.

**3te Stelle: DWORD:** Die Fehler Kennnummer.

**5te Stelle: LPTSTR:** Die Stringvariable in der der lesbare Text gespeichert  
 wird. Hierzu muss text in der Form (LPTSTR) &text übergeben werden.

Nachdem beide Funktionen durchgeführt wurden befindet sich der lesbare Text in der Variable text.

Je nach Belieben kann jetzt mit  
 **int WINAPI MessageBox([…])**eine Messagebox geöffnet werden, die dem Nutzer die Fehlernachricht anzeigt.  
Hierzu müssen der Funktion folgende Parameter mitgegeben werden:

**HWND:** Das Superfenster in dem die Nachricht angezeigt werden soll.

**LPCWSTR:** Der Text im Fenster. Hier empfiehlt es sich den Fehlertext zu  
 verwenden.

**LPCWSTR:** Die Überschrift die die Messagebox tragen soll.

**UINT:** Der Typ der Messagebox als Flag. Im Fehlerfall ist es zu empfehlen hier  
 MB\_OK zu verwenden. MB\_OK lässt die Nachricht als Box mit einem  
 OK-Button anzeigen.

Es empfiehlt sich diesen ganzen Ablauf als Funktion auszulagern und jedes Mal, wenn man davon ausgeht dass ein Fehler auftreten kann, aufzurufen.

Das Ganze eignet sich nicht zum Debuggen, denn es gibt einen entscheidenden Unterschied zu z.B. Java:  
In Java werden Exceptions geworfen die auf jeden Fall (wenn nicht vom Programmierer abgefangen) von der VM gefangen und dem Nutzer ausgegeben werden. In Win32 Anwendungen mit C++ besteht das Problem dass diese Fehler zwar auftreten und gespeichert werden aber man sie beim Lauf des Programms nicht „ins Gesicht gedrückt bekommt“ wenn man sie als Entwickler nicht abfragt.

Hier eine kleine Überlegung:  
Nehmen wir mal an dass eine Funktion den Input einer anderen Funktion produziert. Läuft nun die erste Funktion auf einen Fehler speichert sie die Fehlermeldung und gibt einen fehlerhaften Wert zurück.  
Durch diesen fehlerhaften Wert läuft auch die zweite Funktion auf einen Fehler. Dieser wird wieder gespeichert.  
Wenn jetzt der letzte Fehler abgefragt wird, weil festgestellt wurde dass die letzte Funktion auf einen Fehler lief, bekommt man den Fehler der zweiten Funktion.  
Sicher ist es das was man wollte: den letzten Fehler. Dennoch ist es nicht das was man eigentlich benötigt. Benötigt wäre der Fehler der ersten Funktion.

So muss man dann zu debug Zwecken zu jeder Funktion die schief laufen KANN eine Abfrage schreiben. Somit eignet sich diese Technik nicht zum Debuggen sondern „nur“ um Fehler zu Verarbeitungszwecken dem Benutzer anzuzeigen. Zum Beispiel warum ein Server keine Verbindung aufbauen konnte.

Die Empfehlung ist deswegen das Programm nicht wie in Java mit Ausgaben zu debuggen sondern mit Techniken wie Watchlists und ähnlichem.

Diese Bauweise hat im Vergleich zu Java auch einen Vorteil. Während ein Java Programm durch die VM beendet wird weil ein Fehler aufgetreten ist, der wahrscheinlich nicht mal den Fortlauf der Anwendung beeinträchtigt hätte, kann ein Win32 Programm ohne Unterbrechung ausgeführt werden.

Nachdem jetzt beschrieben ist wie ein normales minimales Win32 Programm aussieht muss noch erklärt werden worin der Unterschied zu MMIX-Edit besteht.

**Aufbau der MMIX-Edit IDE**

**Zusätzliche Hauptstruktur:**

Um interne verarbeitende Funktionen des Programms zusammen zu fassen existiert eine spezielle interne Struktur namens app.

Diese Struktur speichert sowohl die Instanz als auch die Handles der Anwendung und stellt spezielle Funktionen zur Verfügung, welche z.B. das Fenster bearbeiten.   
Die gespeicherten Handles sind das Hauptfenster mit dem Menü, das Editorfenster in dem Scintilla läuft und eine Listbox als Fenster in der die Fehler angezeigt werden.  
Außerdem wird die aktuelle Datei in einer Variable gespeichert (als File Pointer).

Die Struktur bietet verschiedene Funktionen zur Manipulation des Editors an.  
Darunter sind auch die Methoden des Speicherns und Ladens von Dateien. Dazu später mehr.

**Zusätzliche Fenster:**

Zusätzlich zum Hauptfenster mussten zwei weitere Fenster erstellt werden.

Ganz offensichtlich ist der Editor ein eigenes Fenster. Dieses wurde mit Hilfe von CreateWindow erstellt.   
Wenn man als ersten Parameter L“Scintilla“ angibt, greift das CreateWindow in die Scintilla.dll und lädt eine Editorkomponente.

In dem Punkt agiert die Scintilla.dll wie eine Standard Windows dll und bietet das Fenster als Windows Fensterkomponente an. Was gemeint ist wird gleich erklärt.

Das zweite Fenster ist die Fehlerliste. Im CreateWindow muss als erster Parameter L“ListBox“ angegeben werden.

Hier sieht man was vorhin gemeint war: Die ListBox ist eine Windows Fensterkomponente die von der Windows dll angeboten wird. Scintilla hingegen wird von der Scintilla dll angeboten. Beide werden aber auf dieselbe Art und Weise angeboten. Um eine Standard Windows Komponente zu laden muss man beim CreateWindow als ersten Parameter dessen Namen angeben.

Zu beachten ist hier dass im Style Bereich der CreateWindow Methode WS\_CHILD mitgegeben wird. Dies bewirkt dass das Fenster als Kindfenster der Hauptanwendung gezeigt wird. Somit ist die Komponente nicht mehr als einzelnes Fenster zu erkennen sondern wird im Hauptfenster angezeigt. Kinderfenster haben keinen Applikationsrahmen. Der Applikationsrahmen ist der Rahmen der jedes Windowsprogramm umgibt. Er beherbergt die Minimieren-, Maximieren- und Schließenbutton, sowie die Möglichkeit das Fenster mit einem Rand in der Größe zu ändern. Er reagiert unter Windows 7 auch auf die „Aero Snap“ Funktionalität und alles was sich visuell in ihm befindet wird durch „Aero Peek“ angezeigt.

„Aero Snap“ ist die Funktionalität das Fenster, an den oberen/rechten/linken Rand des Bildschirms, zu schieben, sodass sich das es sich von allein, auf die Größe des Bildschirms oder eben die Hälfte dessen, resized.

„Aero Peek“ ist die Windowsfunktion Fenster, bei langem Rasten des Mauszeigers auf einem Windowsleistenelement, in Miniatur an zu zeigen.

Bei der Fehlerliste muss LBS\_NOTIFY ebenfalls als Style angegeben werden damit das Fenster das Hauptfenster darüber benachrichtigt wenn der Nutzer einen Klick/Doppelklick/Rechtsklick auf ein Element in der Liste gemacht hat.

Nützlich ist es den WS\_EX\_ACCEPTFILES bei den Styles des Editor Fensters einzubauen. Dieses Flag bewirkt dass die Editorkomponente durch Drag-And-Drop gezogene Dateien akzeptiert, sprich der Editor lädt das File wenn es in diesen gezogen wird.  
Dieses Drag-And-Drop Ereignis wirft allerdings wieder eine Nachricht die durch den Programmierer abgefangen werden muss.

Auf jeden Fall muss noch das Hauptfenster als Superfenster und die Instanz des Programms weitergegeben werden.

Analog zum Hauptfenster ist es hier erstrebenswert nachzuprüfen ob die beiden Fenster wirklich erstellt wurden.  
Dies macht man wieder mit Hilfe der Technik die im Bereich „Fehlerbehandlung“ beschrieben wurde.

**Vereinfachung der SendMessage Funktion:**

Da die Funktion SendMessage, beim Setzen des Styles der Editorkomponente, sehr oft verwendet wird existiert eine Kurzfassung der Methode speziell für den Editor.

Zur Erinnerung: die SendMessage Funktion benötigt folgende Angaben: ein Handle zum Fenster, die Nachrichtennummer, einen WPARAM und einen LPARAM.  
Die neue Funktion SendEditor benötigt nur noch den Nachrichtencode, einen WPARAM und einen LPARAM. Der WPARAM und der LPARAM sind allerdings defaulted, also mit einem Default-Wert vorbelegt.   
Das hat den Vorteil dass man nicht mehr drauf achten muss was man als WPARAM und was man als LPARAM mit gibt. Die Scintilla Nachrichten hatten ja die Eigenschaft dass sie von 0 bis 2 Parameter benötigen. Nicht benötigte Parameter mussten 0 sein. Diese Bauart der SendEditor-Funktion übernimmt die Aufgabe die restlichen Parameter mit 0 zu füllen.  
Im Grunde wird innerhalb der Funktion nichts anderes gemacht als SendMessage auf dem Editorfenster mit den angegebenen Parametern aufzurufen.

**Laden, Speichern und neu Erstellen einer Datei:**

Da Scintilla keine Dateien laden oder speichern kann mussten Funktionen geschrieben werden die diese Aufgabe übernehmen.  
Es existiert eine New Funktion welche ein neues Dokument öffnet. Dazu muss allerdings zuerst gespeichert werden, wenn nicht schon gespeichert wurde. Danach wird der Editor gelehrt und der Buffer welcher die letzten Schritte merkt gelöscht, um dann einen neuen Dateipfad zu setzen und das Fenster umzubenennen, damit der Dateiname im Titel steht. Der Dateipfad wird auf eine Datei mit dem Namen „New.mms“ gesetzt.  
Die Funktion zum Öffnen von Dateien lehrt den Editor, lädt die Datei (sofern sie existiert) in die zugehörige Variable und setzt danach den Titel des Fensters. Kann die Datei nicht geladen werden gibt das Programm eine Fehlermeldung in Form einer MessageBox aus. Am Schluss wird ein Savepoint gesetzt, der Fokus auf den Editor geschoben und das Caret an die erste Stelle im Dokument gesetzt. Eine Anmerkung zum Laden der Datei: Beim Laden wird im Grunde nichts anderes getan als den Editor zurück zu setzen und den Inhalt der neuen Datei zu übertragen. Dabei wird tatsächlich New (Funktion für neue Datei) aufgerufen.  
Die Öffnen Funktion macht allerdings nichts anderes als die Datei zu öffnen. Wenn man aber eine Datei öffnen will muss man diese zuerst auswählen. Das passiert in einer separaten Methode. Diese Methode ruft die Funktion  
**BOOL WINAPI GetOpenFileName([…])**auf. Hier wird als einziger Parameter ein OPENFILENAME Konstrukt benötigt. Dieses Konstrukt enthält Daten wie das Superfenster, die Instanz, einen Filter, einen Titel und viele mehr. Das Konstrukt dient zum Starten eines Dialogs zum Abfragen einer Datei die geöffnet werden soll. Hierzu werden natürlich alle Daten gebraucht die auch ein normaler Dialog benötigt. Außerdem wird ein Filter benötigt der hilft die Dateien im Anzeigefenster zu filtern (Beispiel: es sollen nur alle MMIX Dateien angezeigt werden. Dann braucht man einen Filter der alle Dateien ausfiltert die nicht mit .mms oder .mmo enden). Liefert die GetOpenFileName Methode daraufhin eine Datei zurück muss diese nur an die eben geschriebene Öffnen Methode übergeben werden.  
Es existieren mehrere Methoden zum Speichern. Eine „normale“ Save und eine SaveAs. Die Save macht nichts anderes als das derzeitige Dokument mit seinem Namen zu speichern, also das normale Strg + S. Die SaveAs Funktion dient zum Speichern der Datei mit neuem Namen. Beide Saves übergeben das eigentliche Speichern der Datei an eine dritte Methode. Diese verlangt nur den neuen Namen der Datei. D.h. die Save macht nichts anderes als die dritte, SaveFile genannte, Methode mit dem aktuellen Namen der Datei aufzurufen. Die SaveAs macht dort schon ein wenig mehr indem sie die Windows Funktion  
**BOOL WINAPI GetSaveFileName([…])**aufruft. GetSaveFileName ist ähnlich zu GetOpenFileName. Sie benötigt ebenfalls ein OPENFILENAME. Im Gegensatz zu GetSaveFileName öffnet sie aber einen Dialog zur Abfrage des Speicherplatzes der neuen Datei. Das „Wollen Sie die vorhandene Datei überschreiben?“ wird von GetSaveFileName übernommen.  
In SaveFile wird eine Datei mit dem Namen geöffnet und mit den Daten im Editor gefüllt. Konnte kein Pointer zu der Datei erschaffen werden (konnte sie nicht geöffnet werden) wird eine MessageBox geöffnet die den Fehler zum Benutzer trägt.  
Eine SaveIfUnsure Methode sorgt für das Speichern einer Datei wenn vom Benutzer eine Eingabe kam die das Schließen der Datei zur Folge hat und die Datei noch nicht gespeichert wurde. Im Endeffekt ist diese Methode für den Dialog „Wollen Sie die Datei speichern bevor sie geschlossen wird?“ zuständig. Im Grunde wird lediglich eine MessageBox mit besagter Meldung geöffnet und, wenn der Benutzer „Ja“ ausgewählt hat, die Datei gespeichert.

Beim Öffnen und beim Speichern einer Datei wird die Windows Funktion  
**FILE \*\_wfopen([…])**benötigt. Diese Funktion öffnet einen File Pointer auf eine Datei. Dabei muss der Funktion der Name der Datei und Flags mitgegeben werden. Die Flags dienen zur Zugriffsberechtigung, d.h. wird die Datei beschrieben, wird von ihr gelesen usw.  
Diese Funktion ist in dem Sinne speziell da sie zum Öffnen von vorhandenen Dateien und zum Erstellen von neuen Dateien verwendet werden kann.  
Wird \_wfopen aufgerufen mit dem Namen einer nicht vorhandenen Datei wird diese einfach neu erstellt. Allerdings macht es hier dann keinen Sinn die Datei mit einem read-only Flag zu öffnen (was dann allerding eher das Problem des Entwicklers ist).

**Scintilla’s eigene WndProc:**

Die Scintilla Komponente hat eine eigene WndProc ähnliche Funktion bekommen. In dieser werden Nachrichten abgehandelt die dem Editor bestimmt sind. Durch diese Methode wird die WndProc nicht allzu lang und unübersichtlich. Zu den Nachrichten gehören Menüeingaben wie Öffnen und Speichern einer Datei, Wiederholen einer Aktion oder das Einfügen von Text.

**Aktivierung von Menüelementen:**

Um Menüelemente, in der Hauptmenüleiste des Programms, zu aktivieren, wenn diese verfügbar sind, gibt es die Funktion EnableAMenuItem. Diese ruft intern die Windows Funktion  
**BOOL WINAPI EnableMenuItem([…])**auf. Diese Funktion ändert den Zustand eines Menüelements. Dazu muss man ihr das Menühandle, die ID des Menüelements und Flags mitgeben. Die Flags besagen wie der Zustand des Menüelements geändert wird.  
An das Menühandle kommt man indem man die Methode  
**HMENU WINAPI GetMenu([…])**aufruft. Diese Funktion benötigt nur das Fenster dem das Menü zugeordnet ist.  
Zu beachten ist hierbei dass das Menü als Ressource eingebunden worden sein muss, wie die oben erwähnte ID verrät.

Eine weitere Methode checkt die Menüeinträge und aktiviert oder deaktiviert sie wenn benötigt. Diese so genannte CheckMenu Funktion aktiviert die Buttons für „Rückgängig“, „Wiederholen“ und „Einfügen“ wenn diese Funktionen ausgeführt werden können. Dies wird geprüft indem man SendMessage (oder SendEditor) mit der Nachricht „SCI\_CANUNDO“, „SCI\_CANREDO“ oder „SCI\_CANPASTE“ aufruft.  
Das eigentlich Interessante in der Funktion ist allerdings die Tatsache dass auch der Save-Button aktiviert/deaktiviert wird. Dies geschieht nach Abfrage der isDirty Variable. Diese Variable ist durch MMIX-Edit definiert. Sie besagt ob das aktuelle Dokument gesichert ist oder nicht. In SaveIfUnsure wird isDirty ebenfalls abgefragt. isDirty wird durch verschiedene Methoden gesetzt, darunter ist auch die Notify Funktion. Notify wird von der Nachrichtenschleife aufgerufen. Dazu später mehr.

**Das Stylen des Editors:**

Zum einfachen Setzen eines Styles im Scintilla Editor gibt es eine SetAStyle Methode die mehrere Funktionsaufrufe zusammenfasst.  
Die Methode erwartet eine Stylenummer mit der der Style selbst angesprochen wird also welcher Style verändert werden soll.  
Des Weiteren werden eine Vordergrundfarbe, eine Hintergrundfarbe, eine Schriftgröße und eine Schriftart verlangt. Diese werden dann dem Style zugeordnet.  
Um die Funktion möglichst praktisch zu machen ist sie im Struct mit Defaultwerten ausgestattet. So ist das einzig wirklich Benötigte die Stylenummer und die Vordergrundfarbe.

Zum Setzen der Styles und zum Initialisieren des Editors existiert die InitialiseEditor Funktion. Sie lädt den Lexer, setzt die Rahmen, erstellt zwei Marker und verändert die Styles. Es gibt zwei Rahmen:  
Ein Rahmen zum Anzeigen der Zeilennummer. Das Schwierige hierbei ist die richtige Breite des Rahmens zu finden. Hat man ein Dokument mit 100 Zeilen muss der Rahmen breit genug sein um die Zahl 100 ohne Probleme anzuzeigen. Ist das Dokument aber nur 7 Zeilen lang ist die Breite, die bei 100 benötigt wird, viel zu groß. Der Trick liegt also darin dass der Rahmen groß genug sein muss um alle Zahlen anzuzeigen aber klein genug dass das Editorfeld nicht zu stark beschnitten wird. Da bleibt nur eine Lösung: die Breite dynamisch anzupassen. Dies macht man mit einem Aufruf von SendMessage/SendEditor mit der Nachricht SCI\_TEXTWIDTH. Wenn man diesem Aufruf einen String mitgibt bekommt man die Breite des Textes zurück geliefert. Mit der so erhaltenen Textbreite kann man immer wieder die Breite des Rahmens anpassen. Eine Besonderheit der Nutzung, dieser Art und Weise der Breitenabfrage, ist im Fazit nachzulesen.  
Beim Laden eines neuen Dokuments muss die Breite nachgesetzt werden. Deswegen ist in der Funktion zum Öffnen eine Breitenänderung eingebaut.

**Vom Füllen und Nutzen der Fehlerliste:**

Zuerst muss die Funktionsweise der Fehlerliste beschrieben werden. ListBoxes unter Windows bieten die Möglichkeit mehr zu speichern als nur das was der Nutzer sieht. So kann man einer ListBox nicht nur einen String mitgeben der dann in die Liste eingetragen wird und angezeigt wird sondern auch zusätzliche Daten die im Hintergrund den Strings in der Liste zugeordnet werden.   
In unserem Fall wird der ListBox der Beschreibungstext des Fehlers als String übergeben und die zugehörige Zeilennummer als versteckter Wert.  
Davon bekommt der Nutzer erst etwas mit wenn er einen Doppelklick auf einen Eintrag in der Liste macht. Zu dem Doppelklick mehr wenn die WndProc beschrieben wird, also später.

Nun zum Füllen der Liste. Der Assembler ruft beim Schreiben der Fehler eine Methode auf, die sich addListString nennt.  
Diese Methode muss als „extern „C““ markiert sein damit sie vom Assembler, welcher in C geschrieben ist, angenommen wird.  
Wenn der Assembler die Methode aufruft gibt er ihr den Fehlertext und die Zeilennummer mit.   
Das Hinzufügen von Strings zur ListBox geschieht mit der Nachricht LB\_ADDSTRING. Hierbei muss lediglich der Text mitgegeben werden (allerdings im LPARAM/zweiten Parameter). SendMessage liefert beim Senden dieser Nachricht den Index des Strings, in der Liste, zurück.  
Will man nun dem String einen versteckten Wert mitgeben gibt es die LB\_SETITEMDATA Nachricht. Diese erwartet den Index des Strings und die Daten die hinzugefügt werden sollen.  
Jetzt hat die Listbox einen Eintrag mit dem Fehlertext und der versteckten Zeilennummer.  
Am Schluss werden im Editor, mit SCI\_MARKERADD, zwei Marker eingefügt. Der erste Marker ist ein roter SCI\_MARK\_FULLRECT d.h. ein Rechteck im Symbolrahmen. Der zweite Marker ist ein grauer SCI\_MARK\_BACKGROUND, also ein grauer Balken hinter der Zeile im Text.

Macht der Nutzer einen Doppelklick auf einen Eintrag in der Liste wird durch die WndProc eine Funktion namens positionCaret aufgerufen. Damit der Nutzer sieht in welcher Zeile der Fehler aufgetreten ist muss nun im Editor das Caret an die Stelle gesetzt werden. Dazu benötigt man zuerst die Zeilennummer welche man aus der Liste holen muss.   
Zuerst muss abgefragt werden welche Zeile, in der Fehlerliste, ausgewählt ist. Dies geschieht mit der Nachricht LB\_GETCURSEL. Um nun aus dem ausgewählten Element wieder die versteckten Daten zu erhalten gibt es die Nachricht LB\_GETITEMDATA die den Index des Elements erwartet.  
Nun kann man einfach mit SCI\_GOTOLINE an die Position im Text gehen.  
Damit das der Nutzer auch mitbekommt wird der Fokus in der Anwendung auf den Editor gelegt, denn er lag dank dem Doppelklick vorher auf der Fehlerliste.

**Externe zur MMIX Integration:**

Um den Assembler sowie den Simulator zu integrieren mussten zwei externe Funktionen und eine externe Variable geschaffen werden.

Da gibt es die Funktionen mmixal und mmixsim. Beide sind mit EXTERN\_C markiert. Dabei handelt es sich lediglich um Prototypfunktionen um die ehemaligen main Methoden des Assemblers und des Simulators einzubinden. Sie enthalten keinen Code da dieser in den C Dateien der beiden MMIX-Komponenten vorhanden ist.

Eine Variable die im Simulator eine Rolle spielt ist ebenso mit EXTERN\_C eingebunden. Dabei handelt es sich um die „interacting“ Variable. „interacting“ gibt an ob der Simulator im interaktiven Modus laufen soll. Beim interaktiven Modus kann man dem Simulator live Befehle geben. Um eben diesen Modus später ein- und auszuschalten muss man irgendwo speichern ob der Modus verwendet werden soll. Man könnte dem Simulator beim Starten diesen Wert mitgeben da man aber sowieso eine Variable dafür erschaffen hat kann man sie gleich zwischen den Programmkomponenten teilen.

**Interne Kommandozeile des Simulators:**

Wie im MMIX Teil beschrieben hat der Simulator eine eigene Kommandozeile. Was nicht beschrieben wurde ist dass diese Konsole in einem eigenen Thread läuft. Um einen Thread dafür zu erstellen gibt es  
**HANDLE WINAPI CreateThread([…])**was einen Thread aus den angegebenen Parametern erstellt und startet.  
Unter den Parametern muss man neben sowas wie Sicherheitsattributen und der Stackgröße auch die Hauptroutine des Threads mitgeben.  
Diese Hauptroutine ist die Main Funktion des Threads. Dem entsprechend muss diese Methode auch analog zu der WinMain geschrieben werden.  
Diese Hauptroutine sieht in etwa so aus:  
**DWORD WINAPI ThreadMain([…])**Der Name kann geändert werden Hauptsache man gibt den richtigen bei CreateThread an.   
Die Hauptroutine ist das Erste das im Thread aufgerufen wird. Dem entsprechend muss alles was der Thread tun muss von dieser Funktion aufgerufen werden.

Die Kommandozeile wird im Thread mit einer speziellen Funktion aufgerufen:  
**BOOL WINAPI AllocConsole()**Diese Funktion startet eine neue Konsole. Diese Konsole unterscheidet stark von der die Windows anbietet. Diese Konsole ist wie bereits beschrieben nur eine schwarze Textbox. Alle Eingaben müssen vom Entwickler abgefangen und verarbeitet werden. Die Ausgabe muss dem entsprechend auch auf die Konsole wieder zurück geleitet werden.

Bei der Integration des Simulators wird nichts anderes gemacht als den Simulator ohne Parameter zu starten und seinen Output auf die Konsole zu leiten.  
Dabei müssen 3 Streams verändert werden: stdout und stderr müssen auf den Output und den Error Output der Konsole gelenkt werden und der Input der Konsole muss auf stdin gestreamt werden. Letzteres muss nicht gemacht werden da nicht davon ausgegangen wird dass der Nutzer eine Eingabe tätigt.

Man sollte den Konsolenbuffer in der Größe ändern, bevor man die Streams piped. Um z.B. die Höhe zu verändern gibt es die Methoden  
**BOOL WINAPI GetConsoleScreenBufferInfo([…])**und  
**BOOL WINAPI SetConsoleScreenBufferSize([…])**welche Informationen über den Buffer holen und setzen.  
Um mit den Methoden zu arbeiten benötigt es eine Struktur namens CONSOLE\_SCREEN\_BUFFER\_INFO, welche die Informationen des Buffers speichert.  
Um nun den Buffer in der Größe festzulegen werden zuerst mit dem oben genannten Getter die Bufferinformationen des Buffers geholt. Danach kann man in der Struktur in der Variable dwSize den Y Wert setzen (bufferInfo.dwSize.Y = …). Zum Schluss setzt man die Buffergröße indem man mit dem Setter die Variable dwSize der Bufferinfo zurückschreibt. Beide Funktionen benötigen als ersten Parameter einen Handle auf den Konsolenstream.Der zweite Parameter ist beim Getter der Verweis auf die Infostruktur und beim Setter die dwSize Variable.   
Den Handle auf den Konsolenstream erhält man mit  
**HANDLE WINAPI GetStdHandle([…])**Der einzige Parameter ist ein DWORD mit der ID des Streams. Die IDs sind STD\_OUTPUT\_HANDLE, STD\_ERROR\_HANDLE und STD\_INPUT\_HANDLE.

Um einen Konsolenstream (richtig wäre File Pointer, aber aus Einfachheit nur Stream) mit einem Standard C/C++ Stream zu verlinken muss der File Pointer welcher zum Beschreiben des C/C++ Streams verwendet wird mit einem File Pointer überschrieben werden der in die Konsole verweist. Um einen solchen File Pointer zu erhalten gibt es die Funktion  
**FILE \*\_fdopen([…])**welche aus einem int und einem Zugangsmodus Flag einen File Pointer erstellt. Zuvor muss aber aus dem Konsolenstream das oben genannte int erstellt worden sein. Dies geschieht mit  
**int \_open\_osfhandle([…])**Hier wird zum einen ein Handle auf einen Konsolenstream benötigt und zum anderen wieder Flags die den Zugang regulieren.  
Als Handle auf den Konsolenstream kann man das durch GetStdHandle erstellte Handle verwenden. Zu beachten ist hierbei dass man dieses vorher auf (long) casten sollte. Warum dies so ist steht im Fazit.

Dieses Rezept zum verlinken der Streams kann man analog auf alle drei Streams anwenden.

Um das Verbinden zu besiegeln muss noch  
**static bool sync\_with\_stdio([…])**aufgerufen werden. Hierbei wird dafür gesorgt dass die Streams und die C/C++ Methoden nicht desynchronisieren.

Nachdem die Streams gekreuzt wurden muss nur noch die mmixsim Methode aufgerufen werden. SO läuft dann mmixsim in einem eigenen Thread und der Output wird auf die neue eigene Konsole gepiped.

**Das Aufrufen der MMIX-Komponenten:**

Für das Assemblieren und das Ausführen des MMIX-Codes existieren zwei Methoden.

Die assemble Methode säubert zuerst die Oberfläche von allen möglichen vorherigen Assemblierungen. So werden mit SCI\_MARKERDELETEALL alle Marker im Editor gelöscht und mit LB\_RESETCONTENT werden in der ListBox alle Einträge entfernt.  
Nach dem Aufräumen wird die editierte Datei mit dem read-only Flag geöffnet. Konnte die Datei geladen werden wird mmixal aufgerufen, konnte die Datei nicht geladen werden wird eine MessageBox mit einer entsprechenden Meldung an den Benutzer ausgegeben.

Die run Methode, welche den Simulator startet, muss die Oberfläche nicht säubern weswegen sie gleich mit dem read-only laden der Datei anfängt. Wie bei assemble startet run den Thread mit der Konsole wenn die Datei geladen werden konnte. Gibt das Laden der Datei nichts zurück wird wieder eine Fehlermeldung mit Hilfe einer MessageBox ausgegeben.

**Die spezialisierte WndProc:**

Hier die Nachrichten die in der WndProc aktiv abgefangen werden:

**WM\_SIZE:** Wird das Fenster in der Größe verändert, wird diese Nachricht geworfen. Hier wird das Fenster intern resized, d.h. der Editor wird auf 4/5 der Höhe des Fensters gesetzt und die ListBox bekommt das restliche Fünftel des Platzes. Damit ist das Fenster intern fest aufgeteilt.

**WM\_COMMAND:** betätigt der Nutzer eine Komponente innerhalb der Anwendung wird die Nachricht WM\_COMMAND abgesetzt. Im Low Word des WPARAM befindet sich die eigentliche Nachricht der Komponente, die betätigt wurde. So werden folgende Unternachrichten abgefangen:

- **IDD\_MENU\_COMPILE:** Das ist die Nachricht des Menüs dass der Assemble   
 Knopf gedrückt wurde. Es wird zuerst SaveIfUnsure und dann   
 assemble aufgerufen. Also zuerst gesichert und dann assembliert.

- **0:** So lustig das jetzt auch klingen mag, es wird tatsächlich die 0 abgefangen.  
 Wenn der Benutzer einen Doppelklick auf ein Fehlerlistenelement  
 macht wird im WPARAM das High Word (obere Hälfte des WPARAM)  
 auf LBN\_DBLCLK gesetzt. Da wir in der WndProc nach WM\_COMMAND aber nur noch das Low Word (untere Hälfte) abfragen  
 muss hier 0 abgefangen werden.  
 => Doppelklick auf Listeneintrag: positionCaret wird aufgerufen, also   
 der Caret an die richtige Stelle im Code verschoben.

- **IDD\_MENU\_RUN:** Diese Nachricht kommt vom Run Button. Es wird zuerst  
 gesichert (SaveIfUnsure), dann assembliert (assemble) und dann   
 schließlich der Code ausgeführt (run).

- **IDM\_ABOUT:** Der „Über“ Button im Menü sendet diese Nachricht. Hier wird   
 ein neuer Dialog aus einer Ressource erstellt. Diese Ressource  
 beschreibt eine MessageBox mit dem Impressum als Inhalt. Als  
 Hauptroutine des Dialogs wird eine About Methode mitgegeben. Diese   
 wird später erklärt.

**WM\_NOTIFY:** Scintilla wirft diese Nachricht wenn sich das Dokument geändert hat.   
Im Grunde wird hier nichts anderes gemacht als die Notify Methode mit dem LPARAM als Parameter aufzurufen. In der Notify Methode wird dank dem LPARAM entschieden ob das isDirty gesetzt wird oder nicht um danach die Menüs checken zu lassen.

**WM\_MENUSELECT:** Wenn der Nutzer ein Menüelement auswählt wird diese Nachricht ausgelöst.   
Im Gegensatz zu WM\_COMMAND wird diese Nachricht tatsächlich von Windows abgesetzt. WM\_COMMAND wird durch die ausgewählte Komponente geworfen.  
Wenn nun diese Nachricht gefangen wird, wird nichts anderes gemacht als die Menüs gecheckt, ob noch alle Menüeinträge richtig aktiviert sind.

**WM\_CLOSE:** Wenn das Programm durch den X-Button geschlossen wird kommt diese Nachricht auf. Zuerst wird hier gesichert und dann das Fenster zerstört.

**WM\_CTLCOLORLISTBOX:** Das wird benötigt um die ListBox zu färben. An einem bestimmten Punkt bei der Erstellung der ListBox wird diese Nachricht abgesetzt. Hierbei erwartet die Komponente dass die ListBox dadurch gefärbt wird, wie der Name schon sagt.  
Zuerst wird gecheckt ob die Liste tatsächlich die Fehlerliste ist. Dies kann man mit Hilfe des LPARAM in dem sich ein Handle auf das Fenster befindet, welches die Nachricht abgesetzt hat.  
Danach wird mit SetBkColor als erstes die Hintergrundfarbe gesetzt und dann mit SetTextColor die Farbe des Textes.  
Das Abfangen dieser Nachricht ist keine Pflicht man sollte es nur tun wenn man eine ListBox einfärben will. Am Ende muss die Nachricht noch einen so genannten Brush zurückgeben mit dem der Hintergrund der ListBox gezeichnet wird.  
Anmerkung: SetBkColor setzt die Hintergrundfarbe des Textes. Diese Hintergrundfarbe befindet sich über dem Backgroundbrush. D.h. wenn man die Hintergrundfarbe des Textes auf blau ändert hat man Text mit blauem Hintergrund auf einem z.B. schwarzen Haupthintergrund. Mit SetBkColor wird nur direkt hinter einem Zeichen eingefärbt.  
Die ListBox in MMIX-Edit hat einen schwarzen Texthintergrund, einen schwarzen Haupthintergrund und weiße Schrift. Damit soll der Stil einer Kommandozeile getroffen werden.

**Die Hauptroutine des „Über“-Fensters:**

Dialogen kann man bei der Erzeugung ebenfalls eine Hauptroutine mitgeben. Dabei handelt es sich weniger um ein WinMain sondern mehr um ein WndProc.  
In der About werden folgende Nachrichten abgefangen:

**WM\_INITDIALOG:** Wird aufgerufen wenn der Dialog initialisiert wird. Muss lediglich true zurückgeben wenn nichts Spezielles wie Variablen initialisiert werden müssen.

**WM\_COMMAND:** Wird benötigt für den OK-Button. Ist dieser gedrückt wird das Fenster einfach geschlossen.

Mehr muss im About nicht gemacht werden.  
Wichtig ist hierbei das Abfangen des OK-Buttons. Ansonsten wird das Fenster nicht geschlossen.