**Aufbau der Anwendung**

**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.

**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:

**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.

**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.

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

**Das Erstellen von Dialogen aus Resourcen:**

Hat man innere Fenster, die durch Resources 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 Resource 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.

**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 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.

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.

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