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. 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.

Das Erstellen von Dialogen aus Ressourcen:

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.

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.

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

Speichern, Laden usw. werden als Nachricht von der Scintilla Komponente geschickt. D.h. Der Auslöser kommt in der WndProc an und muss von dem Programmierer verarbeitet werden.
Um das zu verarbeiten existiert in der Scintilla Struktur eine Methode zum Auflösen der Nachrichten.

Zur Vereinfachung existiert auch in SendEditor Methode welche nichts anderes übernimmt als ein SendMessage, mit dem Editor Handle als erstem Parameter, aufzurufen.

Um den Editor ordentlich zu initialisieren existiert eine InitialiseEditor Methode. Diese sorgt dafür dass der Editor anfangs die richtige Farbgebung und den richtigen Lexer hat.

Des Weiteren musste eine Methode namens positionCursor geschrieben werden um den Cursor, bei einem Doppelklick auf ein Element in der Fehlerliste, richtig zu positionieren.

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 die Standard Windows dll und bietet das Fenster auf diese Art 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: 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.

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 Methode die im Bereich „Fehlerbehandlung“ beschrieben wurde.