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.

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\_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.

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.

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.

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.

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.

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.

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.

Hier sind ein paar Messages zum Verändern einer Markierung 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.

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.  
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.

Weitere im Code verwendete Nachrichten sind Folgende:

**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.

Hier ein paar 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.