

List Gen

Manual

Von Karsten Thamm

Wofür benötigt man ListGen

Nehmen wir an man hat eine Enumeration-Liste. Zu dieser Enum-Liste möchte man eine Funktion schreiben, die einen Enumeration-Wert in einen String übersetzt. Weiterhin möchte man einen String (beispielsweise als Keyword) in einen Enumeration-Wert übersetzen.

Fügt man in die Enumeration einen Wert hinzu, so muß dieser noch an zwei weiteren Stellen hinzugefügt werden, damit die Funktionalität konsistent bleibt.

ListGen automatisiert diesen Prozeß und stellt sicher dass derartige Wartungsarbeiten sich auf einen Punkt konzentrieren.

Ein Beispiel:

Wir schreiben einen plattformunabhängigen Exception-Handler. Dieser soll alle bekannten Exceptions kennen, fähig sein diese in plattformunabhängige Codes umzusetzen und einen Beschreibungstext auszugeben. Weiterhin soll dieser um Custom-Exception codes erweitert werden können:

Listen wir hierfür zunächst die bekannten Exceptoins auf:

```
enum glibExceptionType {  
  
    //-----  
    // Section: Special purpose  
    //-----  
  
    // special  
    glibExceptionType_UNKNOWN = -3,  
    glibExceptionType_Reraised,  
    glibExceptionType_MS_VCPP_THROW,  
  
    //-----  
    // Section: Hardware exceptions  
    //-----  
  
    // Memory  
    glibExceptionType_MEM_AccessViolation = 0,  
    glibExceptionType_MEM_MisalignedAccess,  
  
    // Arithmetic  
    glibExceptionType_ARITH_DivisionByZero,  
    glibExceptionType_ARITH_Overflow,  
  
    // Instruction set  
    glibExceptionType_CPU_IllegalInstruction,  
    glibExceptionType_CPU_PrivilegedInstruction,  
  
    // Debugging  
    glibExceptionType_DEBUG_Breakpoint,  
    glibExceptionType_DEBUG_Singlestep // (fehlendes Komma)  
  
};
```

Bei den blauen Hervorhebungen handelt es sich um Sonderfälle, die wir an anderer Stelle noch näher behandeln werden. Diese sind zum einen explizite numerische Wertangaben um den Enum-Counter zu beeinflussen, zum anderen die Sonderbehandlung für das letzte Element, hier beispielsweise ein fehlendes Komma.

Wir benötigen nun eine Funktion die einen angegebenen System-Exception-Wert in eine plattformunabhängige Exception umwandelt:

```
// special
case 0xffffffff : return glibExceptionType_UNKNOWN; break;
case 0xe1000001 : return glibExceptionType_Reraised; break;
case 0xe06d7363 : return glibExceptionType_MS_VCPP_THROW; break;

//-----
// Section: Hardware exceptions
//-----

// Memory
case STATUS_ACCESS_VIOLATION : return glibExceptionType_MEM_AccessViolation; break;
case STATUS_DATATYPE_MISALIGNMENT : return glibExceptionType_MEM_MisalignedAccess; break;

// Arithmetic
case STATUS_INTEGER_DIVIDE_BY_ZERO : return glibExceptionType_ARITH_DivisionByZero; break;
case STATUS_INTEGER_OVERFLOW : return glibExceptionType_ARITH_Overflow; break;

// Instruction set
case STATUS_ILLEGAL_INSTRUCTION : return glibExceptionType_CPU_IllegalInstruction; break;
case STATUS_PRIVILEGED_INSTRUCTION : return glibExceptionType_CPU_PrivilegedInstruction; break;

// Debugging
case STATUS_BREAKPOINT : return glibExceptionType_DEBUG_Breakpoint; break;
case STATUS_SINGLE_STEP : return glibExceptionType_DEBUG_Singlestep; break;
```

Weiterhin benötigen wir eine Funktion die zu einem Exception-Code einen Beschreibungstext ausgibt:

```
switch(eExceptionCode) {

    //$$$$$$%pattern>>>> exceptions toString
    //-----
    // Section: Special purpose
    //-----

    // special
    case glibExceptionType_UNKNOWN: return "unknown exception.";
    case glibExceptionType_Reraised: return "Exception was thrown again after being
handled.";
    case glibExceptionType_MS_VCPP_THROW: return "A Microsoft Visual c++ Exception was thrown
using throw(), but not basically handled.";

    //-----
    // Section: Hardware exceptions
    //-----

    // Memory
    case glibExceptionType_MEM_AccessViolation: return "Memory access violation.";
    case glibExceptionType_MEM_MisalignedAccess: return "Memory Misaligned Access.";

    // Arithmetic
    case glibExceptionType_ARITH_DivisionByZero: return "ARITHMETIC: Division by Zero.";
    case glibExceptionType_ARITH_Overflow: return "ARITHMETIC: Overflow.";

    // Instruction set
    case glibExceptionType_CPU_IllegalInstruction: return "CPU: Illegal instruction.";
    case glibExceptionType_CPU_PrivilegedInstruction: return "CPU: Privileged Instruction.";

    // Debugging
    case glibExceptionType_DEBUG_Breakpoint: return "DEBUG: Breakpoint hit (inline
interrupt ??).";
    case glibExceptionType_DEBUG_Singlestep: return "DEBUG: Single step instruction.";
}
```

Das Hinzufügen eines neuen Exception Code würde in unserem Beispiel bedeuten daß man an 3 verschiedenen Stellen Änderungen vornehmen muß um einen Eintrag hinzuzufügen. Genau in diesem Punkt nimmt einem ListGen sehr viel Arbeit ab. Man denke eventuell an Tastaturcodes, States, Steuerbefehle, etc. die an vielen Stellen Berücksichtigung finden müssen.

ListGen in die IDE einbinden

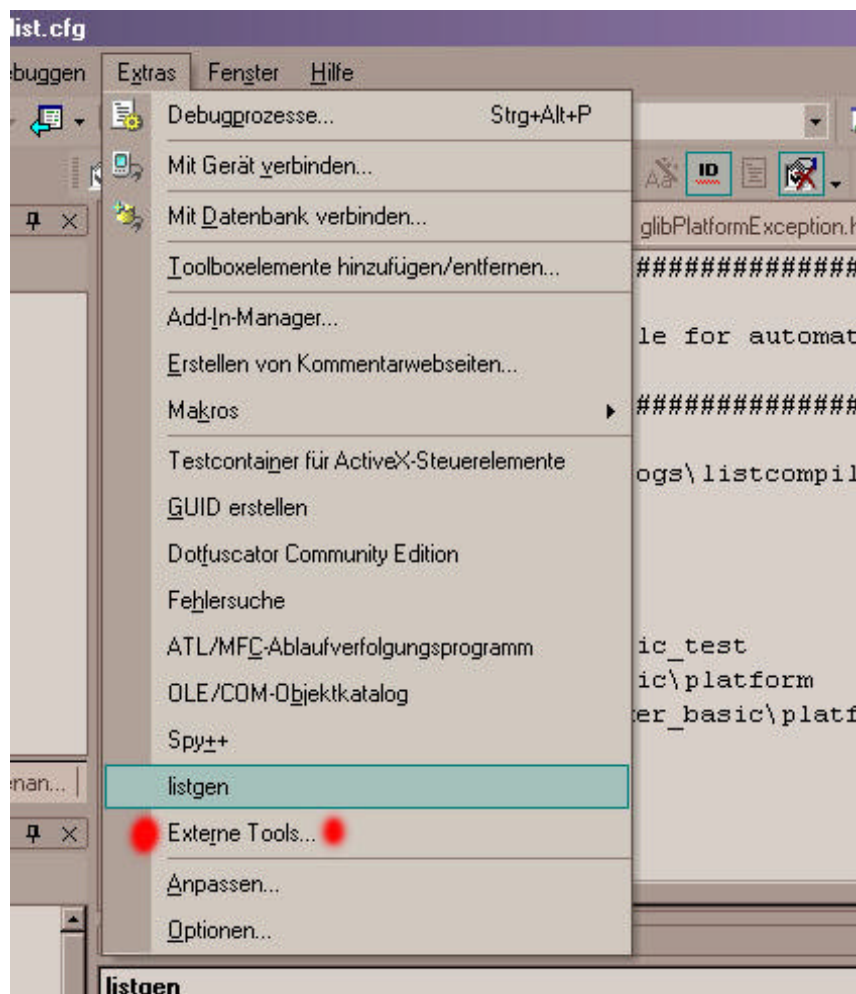
Zunächst möchte ich beschreiben wie man ListGen am sinnvollsten in die IDE einbindet. Dies werde ich am Beispiel VisualStudio 7 verdeutlichen:

ListGen nimmt 3 Parameter:

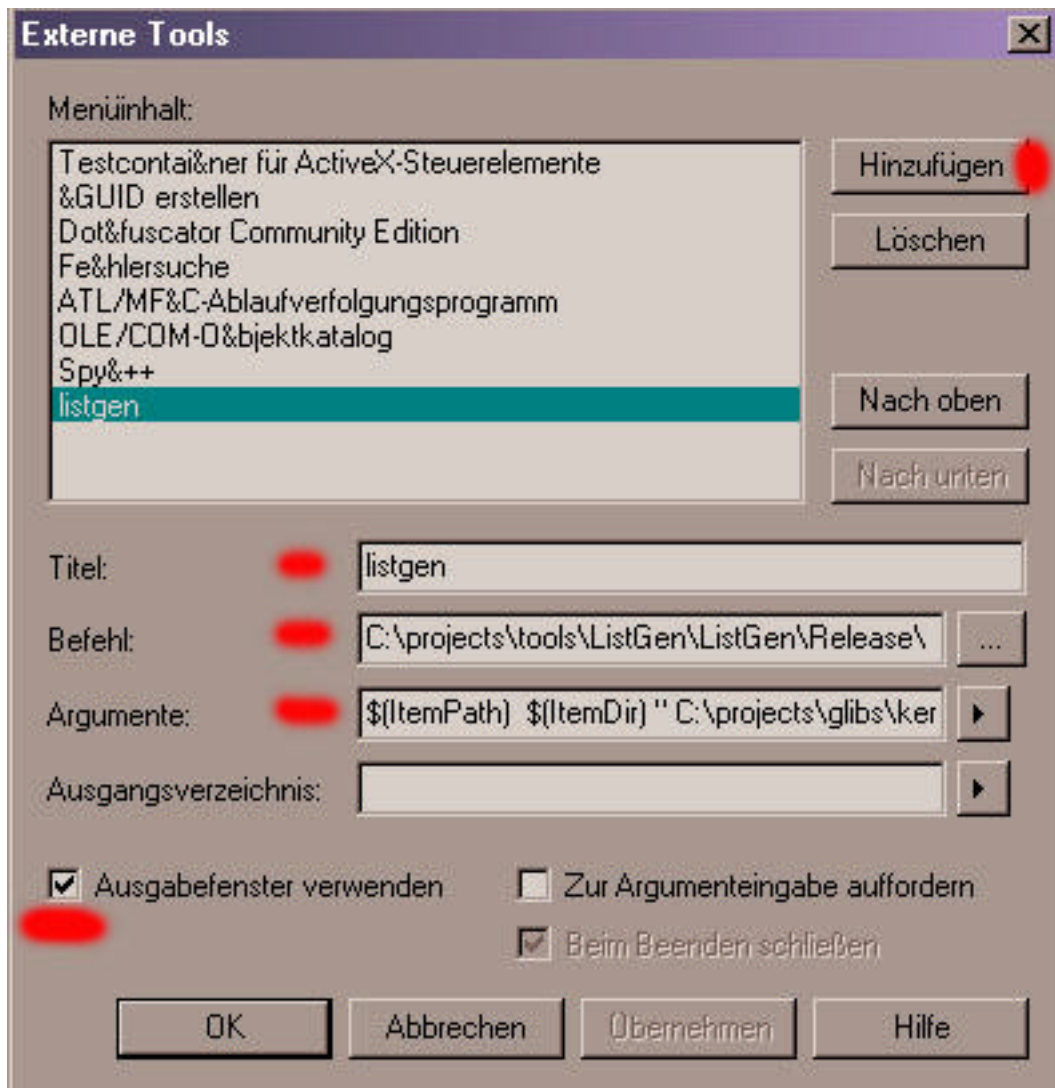
```
ListGen <Quellfile> <Quellpfad> <Configfile>
```

Leider kommt ListGen beim Aufruf nur mit vollen Pfadangaben zurecht. Weiterhin benötigt ListGen ein Maschinenspezifisches Configfile, in dem alle absoluten Pfade enthalten sind.

Micro\$oft VisualStudio kennt sogenannte externe Befehle, mit deren Hilfe beliebige Kommandozeilentools aufgerufen werden können.



Ein Klick auf den ‚Externe Tools‘ (External Tools) – Menüpunkt öffnet die Oberfläche zum Hinzufügen von externen Tools:



Als Titel kann irgend ein beschreibender Text angegeben werden. Unter diesem Namen erscheint dann ‚listgen‘ im Toolmenü.

Der ‚Befehl‘ ist der Pfad von ListGen auf der Platte.

Als Argumente sollte folgendes verwendet werden:

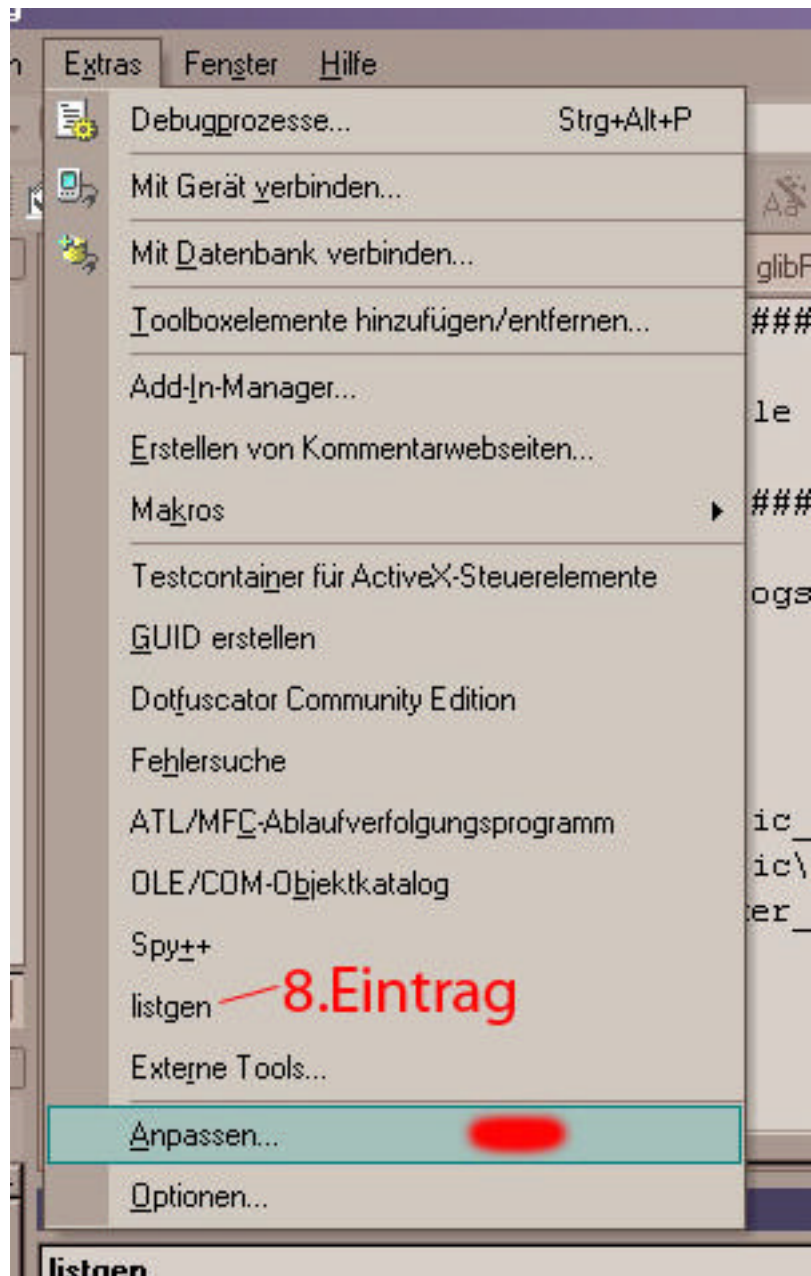
```
$(ItemPath) $(ItemDir) " <Pfad zum Configfile>
```

Beispiel:

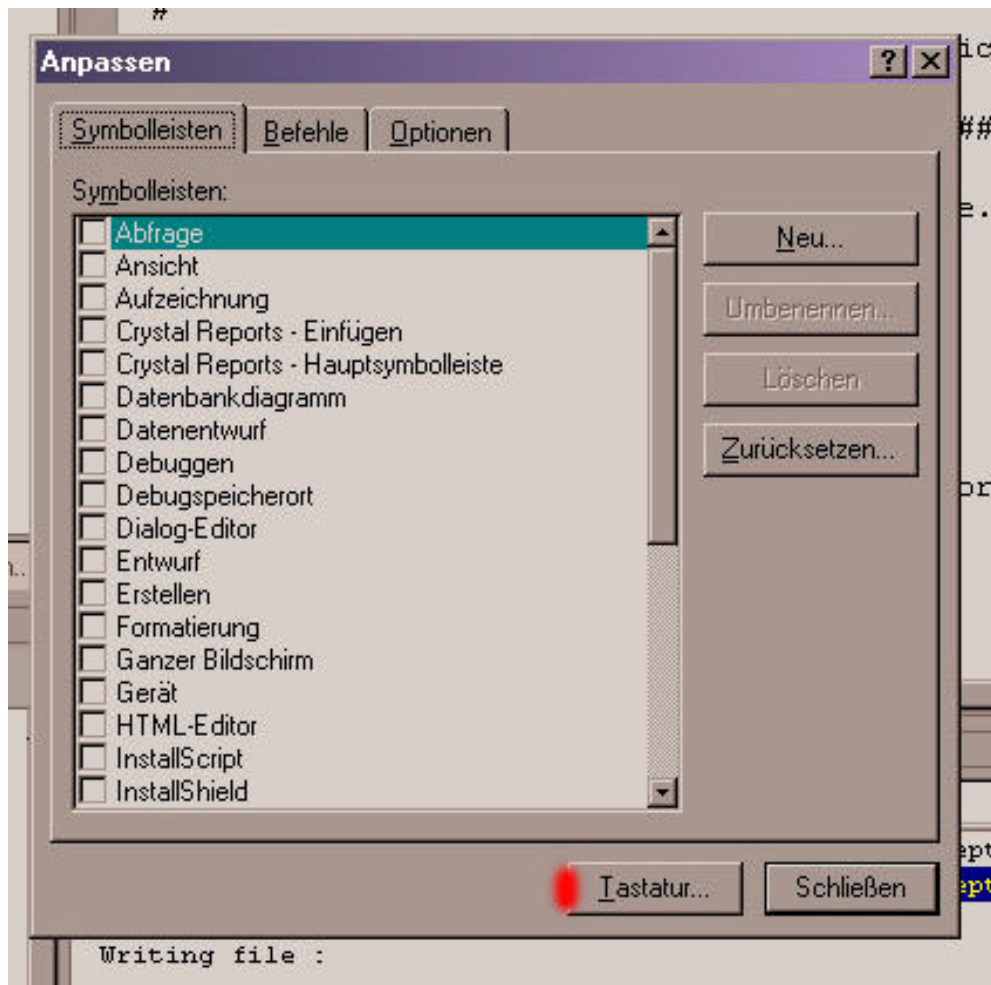
```
$(ItemPath) $(ItemDir) " C:\projects\glibs\ker_basic\codelist.cfg
```

Wozu das Gänsefüßchen dienen soll weiß ich auch nicht so genau, aber bei mir funktioniert es nur mit diesem.

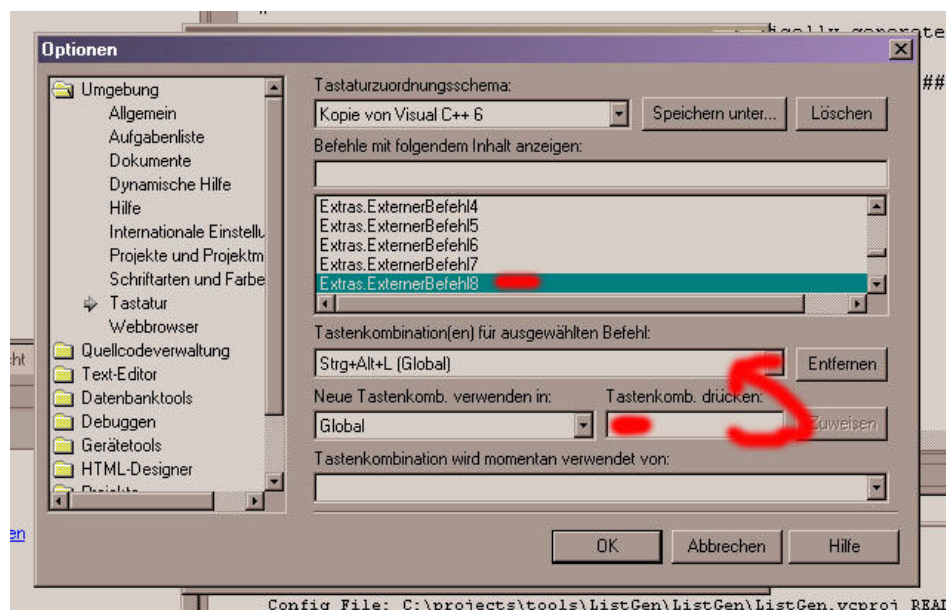
Nun kann man sich zusätzlich noch einen Tastaturshortcut definieren. Hierzu sollte man allerdings zunächst mal zählen um den wievielten Eintrag es sich bei Listegen handelt:



ListGen steht also hier bei den Tools an 8.Stelle Das ist aber bei jedem individuell verschieden. Unter dem Menüpunkt ,anpassen' erhalten wir folgendes Fenster:



Mit einem Klick auf ‚Tastatur‘ erhält man das Shortcut-Konfigurationsmodul:



Man suche sich den richtigen ‚Befehl‘ (in unserem Falle **Extras.ExternerBefehl8**), gebe den Shortcut ein und klicke auf ‚Zuweisen‘. Somit lässt sich Listgen dann elegant per Shortcut (oder per Menü für die Klick- and Point Enthusiasten) aufrufen.

Das Config-File

Das Config-file erfüllt im wesentlichen eine einzige wichtige Aufgabe: Das Festlegen aller absoluten Pfade. Dieses File muß für jeden Rechner angepasst werden, außer für den Fall dass jeder Entwickler die gleichen Pfade verwendet (was in den meisten Projekten auch der Fall ist, aber nicht sein muß).

```
@logfile: C:\projects\glibs\logs\listcompile.log

# All paths to search into

@begin_paths
    C:\projects\glibs\ker_basic_test
    C:\projects\glibs\ker_basic\platform
    C:\projects\glibs\ker_basic\platform_more
@end_paths
```

Unter ‚@logfile:‘ muß der Pfad für das Logfile angegeben werden. Dies wird zwar in der Regel nicht benötigt, da alle Ausgaben auch in der IDE mitgeloggt werden, aber es wird dennoch angelegt und muß demnach angegeben werden.

Zwischen @begin_paths und @end_paths müssen nun alle Pfade gelistet werden in denen sich potentiell Quelldateien befinden, die von ListGen analysiert oder verändert werden. Die Angabe der einzelnen Dateien erfolgt dann innerhalb eines weiteren Konfigurationsblocks.

Der Pfad der Konfigurationsdatei ist der dritte Kommandozeilenparameter beim Aufruf von ListGen (siehe letztes Kapitel).

Inline-Anweisungen

Alle über die Grundkonfiguration hinausgehenden Angaben können bei ListGen entweder inline in Quellcode-Kommentaren vorgenommen werden oder in gesonderten Listen-Dateien. Letzteres ist für die meisten Fälle zu empfehlen, da Listendefinitionen den Quellcode aufblähen und riesige, unschöne Kommentarblöcke zur Folge haben.

Inline-Anweisungen werden immer zwischen zwei Listgen-Tags eingerahmt. Alle außerhalb dieser Tags liegenden Texte werden ignoriert. Diese Listgen-Tags haben immer folgendes Aussehen:

```
$$$$$Blocktyp>>> Parameterliste
```

zum Einleiten eines Listgen-Blocks

```
$$$$$Blocktyp<<<
```

zum Abschließen einer Parameterliste. Beim Parsen von Dateien sind diese Anweisungen der Hinweis für ListGen dass es sich bei den dazwischenliegenden Zeilen um ListGen-Anweisungen handelt.

ListGen Blocktypen

Es gibt drei verschiedene Listgen Blocktypen:

- Config-Blöcke
- Listen-Definitionen
- Pattern-Blöcke

Config-Blöcke sind Blöcke in denen Listen deklariert und weitere Konfigurationen vorgenommen werden können.

In Listen-Definitionen finden sich dann die eigentlichen Wartungsbereiche, in denen die Listen definiert und gewartet werden können.

Pattern-Blöcke sind die Bereiche die dann letztendlich ersetzt werden. In einem Pattern-Block darf sich kein wichtiger Quelltext befinden, da dieser beim Ausführen von ListGen überschrieben wird.

Der Config-Block

Config-Blöcke können in beliebiger Anzahl vorkommen. Sie werden in aller Regel ignoriert, jedoch nicht für die Datei die beim Aufruf von ListGen als erster Parameter übergeben wird. Nur die config-Blöcke dieser Datei werden von ListGen gelesen und zwar vor allen anderen Blöcken. Wenn sich also mehrere config-Blöcke in einer Datei befinden, dann werden zunächst alle Config-Blöcke gelesen und verarbeitet bevor Listgen damit beginnt seine eigentliche Arbeit zu verrichten.

```
$$$$$$%conf>>>> lists.conf

@begin_list
    exceptions
@end_list

@begin_files
    glibPlatformException.h
    glibPlatformException.cpp
@end_files

@begin_patterns: exceptions

    @pattern:      norm   enum           1      $$ ,
    @pattern:      last   enum           1      $$
    @pattern:      n_norm enum           1      $$ = # ,
    @pattern:      n_last enum           1      $$ = #

    @pattern:      norm   win2glib        1      $case @1: return $; break;
    @pattern:      last   win2glib        1      $case @1: return $; break;
    @pattern:      n_norm win2glib        1      $case @1: return $; break;
    @pattern:      n_last win2glib        1      $case @1: return $; break;

    @pattern:      norm   glib2win        1      $case $: return @1; break;
    @pattern:      last   glib2win        1      $case $: return @1; break;
    @pattern:      n_norm glib2win        1      $case $: return @1; break;
    @pattern:      n_last glib2win        1      $case $: return @1; break;

    @pattern:      norm   tostring        1      $case $:return "@0";
    @pattern:      last   tostring        1      $case $:return "@0";
    @pattern:      n_norm tostring        1      $case $:return "@0";
    @pattern:      n_last tostring        1      $case $:return "@0";

@end_patterns
```

```
$$$%$$$%conf<<<<
```

Ein config-Block besteht normalerweise aus drei Bereichen:

- Eine Listendeklaration
- Eine Filedeklaration
- Die Pattern-Deklaration

Die Listen-Deklaration erfolgt innerhalb von @begin_list/@end_list tags. Sie enthält durch Zeilenumbruch getrennt die Namen aller zu verarbeitenden Listen. Taucht später beim Bearbeiten der Dateien eine Liste auf, die nicht in dieser Deklaration enthalten ist, dann wird diese ignoriert. Somit ist es möglich gemischte Listen selektiv zu verändern.

Die File-Deklaration enthält die Liste an Files (ohne Pfade !!) die bearbeitet werden. Diese files werden im Suchpfad gesucht, der im Config-File angegeben ist.

Pattern Deklarationen

Zwischen @begin_patterns und @end_patterns steht dann die eigentliche Pattern-Deklaration für die Liste. Hierbei handelt es sich um eine Schablone wie die Liste generiert wird. Mit @pattern wird ein einzelner Patterneintrag eingefügt.

@begin_patterns hat noch einem Parameter. Hierbei handelt es um die Liste für die die Pattern-Deklaration erfolgt.

Patternbeschreibung

Innerhalb des Patternblocks erfolgt dann die Pattern-Beschreibung. Jede Liste kann eine beliebige Anzahl von Patterns haben. Jedes Pattern erhält einen Namen und benötigt vier beschreibende Einträge mit dem folgenden Format:

```
@pattern:    <Typ> <Name>          <Indent>    $<pattern>
```

Der Typ kann hierbei einer der folgenden Bezeichner sein:

- norm
- last
- n_norm
- n_last

„Norm“ legt fest dass es sich um das „normale“ Pattern handelt. Dies wird im Normalfall bei der Listengenerierung verwendet. „Last“ enthält das Muster für das Letzte Element. Dies ist beispielsweise bei Enumerations nützlich um für den letzten Eintrag das Trennungskomma zu entfernen.

Die Typen mit vorgestelltem „n_“ legen fest dass es sich um „Nummerierte“ Patterns handelt. Ob ein Listeneintrag nummeriert oder nicht nummeriert ist kann bei der Listendefinition festgelegt werden.

Achtung !! Jedes Pattern MUSS für alle vier Typen definiert sein, auch wenn es faktisch zwischen diesen Typen keine Unterscheidung gibt. In diesem Fall muß eine redundante Definition erfolgen.

Patterns haben einen Namen. Dieser muß hier festgelegt werden.

Der Indent gibt die Anzahl der Tabspace an die vor der Pattern-Substitution erfolgen. Der generierte Text soll sich schließlich optisch ansprechend in den Quellcode einpassen. Der Indent Parameter für den ‚norm‘-Typ gibt zusätzlich an wie weit die generierten Kommentare eingerückt werden.

Zum Schluß erfolgt die Pattern-Definition, die mit einem \$-Zeichen eingeleitet wird. List-Gen fügt exakt diesen Text in den Quellcode ein, ersetzt jedoch vorher einige Symbole:

§	wird ersetzt mit dem ‚Namen‘ des Listeneintrags
#	wird ersetzt mit dem numerischen Zählerwert des Listeneintrags
@0 - @9	wird ersetzt mit einem optionalen Substitutionsparameter (max. 10)

Geplant sind noch weitere Substitutionssequenzen, **die aber in der derzeitigen Version nicht enthalten sind (Stand 17.8.2005) (Nachtrag: Jetzt isses DRINNNN):**

\#	wird ersetzt zu ‚#‘
\\$	wird ersetzt zu ‚\$‘
\@	wird ersetzt zu ‚@‘
\\$	wird ersetzt zu ‚\$‘
\n	wird ersetzt zu <Zeilenumbruch>
\t	wird ersetzt zu <Tabspace>
\i	wird ersetzt zu Tabspace, und zwar genau der Anzahl des indent Wertes
\\	wird ersetzt zu ‚\‘

Abgeschlossen wird eine Pattern-Definition mit dem Ende der Zeile.

Eine Pattern-Deklaration besitzt dann immer vier Zeilen, die folgendermaßen aussehen können:

```
@pattern:      norm  enum      1      $$ ,
@pattern:      last  enum      1      $$
@pattern:      n_norm enum      1      $$ = # ,
@pattern:      n_last enum      1      $$ = #
```

Es ist wichtig zu erwähnen daß eine Pattern-Deklaration immer an eine Liste gebunden ist. Man kann keine Pattern-Deklaration für mehrere Listen durchführen. Sollte dies erwünscht sein, dann muß diese Deklaration redundant erfolgen.

Die Pattern-Namen sind im übrigen lokal, d.h. der Namen ‚enum‘ darf für das gleiche Pattern für mehrere Listen verwendet werden.

Commentstyle-Anpassung

ListGen ist eigentlich für C++ Quellcode entwickelt worden. Automatisch generierte Kommentare werden deshalb auch standardmäßig im C++ Stil angegeben. Um auch andere Programmiersprachen unterstützen zu können wurde nachträglich ein Commentstyle-Befehl eingeführt mit dessen Hilfe individuell für jedes Pattern die Einleitungs- und Endsequenz eines Kommentars bestimmt werden kann.

ACHTUNG: Die Deklaration des comment-Styles muß NACH dem Patternblock erfolgen !!

Hier die Syntax:

```
@commentstyle: <pattern-Name> <Einleitungssequenz> <Endsequenz(optional)>
```

Beispiele:

Basic:	@commentstyle: <pattern-Name> rem
Classic C:	@commentstyle: <pattern-Name> /* */
Linux Shell Script:	@commentstyle: <pattern-Name> #
DOS-Batch:	@commentstyle: <pattern-Name> ;
TCL:	@commentstyle: <pattern-Name> #
XML:	@commentstyle: <pattern-Name> <!-- -->

Der Listen-Definitions-Block

Hat Listgen das Einlesen der Configdaten aus der Configdatei und der Hauptdatei beendet, dann beginnt das Programm damit alle in der File-Section angegebenen Dateien (sofern vorhanden) in allen in der config-Datei angegebenen Pfaden zu suchen und bearbeiten.

Hierzu werden zunächst aus allen Files die Listen-Definitionen gelesen.

Listendefinitionen sind immer hierarchisch mit **genau** drei Hierarchieebenen angeordnet. Diese Hierarchie hat keine funktionale, aber eine optisch-strukturelle Bedeutung. Sehen wir uns die Listendeklaration für unsere Exceptions aus dem Beispiel von weiter oben an. Aus Platzgründen musste ich leider den Quelltext etwas kürzen (fehlender Text mit ... angedeutet).

Das volle Beispiel liegt dem Tool bei.

```
$$$$$$%list>>>> exceptions
```

```
$ Special purpose
```

```
§ special
```

```
# -3
glibExceptionType_UNKNOWN    unknown exception.    0xffffffff
glibExceptionType_Reraised    Exception was...    0xe1000001
glibExceptionType_..._THROW    A Microsoft ...handled.0xe06d7363
```

```
$ Hardware exceptions
```

```
§ Memory
```

```
# 0
glibExceptionType_...Vition Memory ... violation.    STATUS_ACCESS_VIOLATION
glibExceptionType_...cess Memory ... Access.        STATUS_DATATYPE_MISALIGNMENT
```

```

§ Arithmetic

    glibExceptionType_...Zero    ARITHMETIC: ...Zero.    STATUS_INTEGER_..._ZERO
    glibExceptionType_...OverflowARITHMETIC: Overflow.STATUS_INTEGER_OVERFLOW

§ Instruction set

    glibExceptionType_...ructionCPU: ... instruction. STATUS_ILLEGAL_INSTRUCTION
    glibExceptionType_...ructionCPU: ... Instruction. STATUS_...INSTRUCTION

§ Debugging

    glibExceptionType_...kpoint DEBUG: ... ??).          STATUS_BREAKPOINT
    glibExceptionType_...estep  DEBUG: ... ruction.       STATUS_SINGLE_STEP

$$$$$$$$list<<<<

```

Die erste Zeile muß hierbei mit einem Section-Token beginnen (\$), gefolgt von einem Section comment (beliebiger Beschreibungstext für die Section).

Dem muß eine Subsection-Deklaration folgen (§).

Danach ist man in der Gestaltung seiner Listendeklaration frei. Ich schlage jedoch vor aus Gründen der Übersichtlichkeit, jede längere Liste in Sections und Subsections aufzuteilen.

Zeilen die Mit \$ Beginnen kennzeichnen also einen Section-Kopf, Zeilen die mit § beginnen kennzeichnen einen Subsection Kopf. Beginnt eine Zeile mit #, bewirkt dies dass der interne Zähler auf den nachfolgenden numerischen Wert gesetzt wird. Weiter hin bewirkt dies dass der nachfolgende Listeneintrag im numerischen Modus weitergeführt wird. Dies bedeutet dass das numerische Pattern (siehe letztes Kapitel) verwendet wird.

Der numerische Mode lässt sich beliebig ein- und ausschalten:

% <counter value> schaltet den numerischen Mode ein und setzt den internen Counter auf den nachfolgenden Wert
<counter value> schaltet den numerischen Mode nur für den nachfolgenden Listeneintrag ein, danach wieder aus und setzt den internen Zähler auf den angegebenen Wert.
+ schaltet den numerischen Modus ein, ohne den Zähler zu verändern
- schaltet den numerischen Modus wieder aus

Jede dieser Numeric-Anweisungen muß in einer eigenen Zeile stehen.

Listeneinträge sind alle nichtleeren Zeilen, die mit einem token (zusammenhängende Zeichenketten ohne Tab) beginnen, das nicht eines der folgenden ist:

§,\$, #, %, -, +

Ein Listeneintrag beginnt mit dem Namen des Eintrags. Dieser kann im Pattern mittels § substituiert werden. Danach folgen mit Tab getrennt die optionalen Parameter, die Leerzeichen enthalten dürfen, aber selbstredend keine Tab-Spaces. Diese optionalen Parameter können dann mittels @0-@9 im Pattern-String substituiert werden. Deshalb darf jeder Listeneintrag maximal 10 Patterns enthalten.

Wird in einem Pattern ein Substitution Tag referenziert dass überhaupt nicht existiert (beispiel: man verwendet @9, aber der Listeneintrag hat nur 3 Optionale Parameter), dann wird der Substitution-Tag durch einen Leerstring ersetzt, also eliminiert.

Pattern-Blöcke

Nun möchte man diese Listen natürlich in den Quellcode einfügen. Hierzu bedarf es keiner großen Anstrengungen. Sofern die Listendeklaration, Pattern-Definition und eine Listendefinition existieren, muß man ListGen einfach nur noch zu sagen an welcher Stelle im Quellcode welche Liste unter Verwendung welches Patterns eingefügt wird.

Für unser obiges Beispiel geschieht dies durch folgende Inline-Tags:

```
//$$$$%%pattern>>> exceptions enum  
//$$$$%%pattern<<<  
  
//$$$$%%pattern>>> exceptions win2glib  
//$$$$%%pattern<<<  
  
//$$$$%%pattern>>> exceptions glib2win  
//$$$$%%pattern<<<  
  
//$$$$%%pattern>>> exceptions tostring  
//$$$$%%pattern<<<
```

Dies teilt ListGen mit dass der Bereich zwischen den Tags mit den entsprechenden Listen gefüllt werden soll. Ein Pattern-Block hat zwei Parameter. Der erste ist der Name der Liste, die deklariert und definiert sein muß. Der zweite ist das Pattern dass für diese Liste verwendet werden soll.

Wichtiger Hinweis:

Bevor ListGen sich ans Werk macht und die wertvollen Quelltexte verändert, wird ein Backup durchgeführt (im gleichen Pfad). Bei einem Absturz von ListGen gehen somit keine Daten verloren, da immer alle Daten in mindestens einem File (Original oder Backup) vorhanden sind.