Inno Setup

Version 2.2.4 (070819)

Über dieses Dokument

Als ich anfing, meine ersten Schritte mit Inno Setup zu unternehmen, war das Tutorial von Johannes Tränkle (http://www.dsdt.info/tutorials/inno/) sehr hilfreich. Ich möchte es daher jedem Anfänger ans Herz legen. Ich wollte aber tiefer einsteigen und stieß daher schnell an die Grenze. Aber das lag nicht an Johannes´ Arbeit. Er hat sich auf die Grundlagen konzentriert, und das ist für ein Einstiegstutorial auch völlig ausreichend.

Auch wenn meine Dokumentation also etwas weiter geht, erhebe ich nicht den Anspruch, eine vollständige, professionelle und in jeder Hinsicht fehlerfreie Anleitung abzuliefern. Ich habe in der Hauptsache die Dinge angesprochen, die mich selbst interessierten.

Ich habe die Dokumentation ein wenig umgestaltet. So ist etwa die Skriptsprache weiter nach vorn gerutscht. Wenn ich Ihnen zum Beispiel in einem Kapitel erläutere, wie Sie in Ihrem Setup so genannte *Tasks* (= Aufgaben) benutzen, dann kann ich Ihnen den dazu passenden Teil der Skriptsprache, mit dem Sie solche Tasks abhängig von bestimmten Bedingungen deaktivieren können, auch im gleichen Kapitel erklären.
So entsteht ein hoffentlich geschlossenes Gesamtbild der Dokumentation. Betrachten Sie die einzelnen Kapitel also bitte nicht als ganz so streng voneinander getrennt.

Mein Dank geht an

Johannes Tränkle
Michael Puff
Markus Fuchs
Thomas Klingler

Inhalt

Grundlagen

Inno Setup ist ein skriptbasiertes Installationsprogramm von Jordan Russell. So ein Skript lässt sich bereits mit einem gewöhnlichen Texteditor erstellen. Einfacher geht es natürlich mit dem Editor von Inno Setup selbst, da der auch das so genannte *Syntax Highlighting* bietet, was die Übersichtlichkeit erhöht und die Arbeit verbessert.

Noch einfacher ist die Arbeit mit einem Hilfsprogramm, das die Syntax des Skriptes komplett vom Anwender fernhält und die Auswahl von Dateien und diverse Einstellungen über Dialoge und Assistenten ermöglicht. Ein Beispiel dafür wäre ISTool, mit dem Sie ein Setup salopp gesagt "zusammenklicken" können. Für diese Dokumentation spielen solche Hilfsprogramme allerdings keine Rolle.

Ich habe mich bemüht, notwendige Befehle und Funktionen möglichst allgemein zu erläutern. Dennoch ließ es sich nicht vermeiden, diverse Anweisungen auch anhand eines Beispiels zu zeigen. Sehen Sie solche Beispiele bitte nur als Mittel zum Zweck. Ich musste mich für einen Dateinamen o.ä. entscheiden. Sie sollten, wenn Sie der Dokumentation Schritt für Schritt folgen möchten, gleich Ihre eigenen Dateien usw. verwenden.

Eine Beispielinstallation

Beginnen wir nun mit einem neuen leeren Inno Setup-Skript.

Dateien angeben

Um Dateien anzugeben, die das Setup später auf den Zielrechner kopieren soll, benötigen Sie die Sektion "`[Files]`". Die Dateien können sich auf einer beliebigen Partition, in einem beliebigen Ordner befinden. Geben Sie in dem Fall bitte einfach den kompletten Pfad mit an. Befinden sich Dateien im gleichen Ordner wie das Skript, bzw. in Ordnern unterhalb des Skriptes, können Sie auf die Pfadangaben verzichten, bzw. relative Pfade benutzen.

Als Beispiel möchte ich Ihnen die Hauptanwendungsdatei zeigen, die sich im gleichen Ordner wie das Setupskript befindet

```
[Files]
Source: "MyProg.exe"; DestDir: "{app}"
```

Unschwer zu erkennen ist der Name der Anwendungsdatei. Damit diese Datei später auch im richtigen Verzeichnis auf dem Zielrechner landet, wurde hier die Konstante "`{app}`" benutzt. Es handelt sich dabei um eine Inno Setup-interne Konstante, die immer auf das vom Anwender ausgewählte Zielverzeichnis zeigt. Üblicherweise handelt es sich dabei um einen Ordner, der sich innerhalb des "`Programme`"-Verzeichnisses Ihres Systems befindet, und der von Ihrem Setup vorgegeben wird. Auf diese Vorgabe komme ich später noch zurück.

Nicht jede Datei soll später aber in dieses Hauptzielverzeichnis kopiert werden. Viele Entwickler nutzen Unterverzeichnisse für Dokumentationen, Quellcodes, Beispiele, usw. Auch das lässt sich relativ einfach realisieren, indem Sie den gewünschten Ordner einfach an die o.g. interne Konstante anhängen. Nehmen wir an, ich möchte mehrere Hilfedateien mit der Dokumentation in den Unterordner "`docs`" kopieren lassen, dann könnte das so aussehen

```
Source: "docs\*.hlp"; DestDir: "{app}\docs"
```

Hier sehen Sie, dass sich die Hilfedateien selbst in einem Unterordner, relativ zum Skript gesehen, befinden. Aus dem Grund muss der Unterordner, der hier auch "`docs`" heißt, bei der Quelle mit angegeben werden.
Das Beispiel zeigt Ihnen übrigens auch, dass Sie mit den gewohnten *Wildcards* aus dem System mehrere Dateien auf einmal in das Setup aufnehmen können. Intern wird für jede Datei ein separater Eintrag angelegt, so dass bei der Deinstallation auch wirklich nur die Dateien entfernt werden, die vom Setup installiert worden sind.

Wollen Sie zum Beispiel alle Dateien inkl. aller vorhandenen Unterverzeichnisse in das Setup aufnehmen, dann genügt das zusätzliche Flag `recursesubdirs`. Im folgenden Beispiel würde der Compiler daher alle Dateien in das Setup aufnehmen, die sich im gleichen Ordner wie das Skript, sowie in allen gefundenen Unterordnern befinden

```
Source: "*"; DestDir: "{app}"; Flags: recursesubdirs
```

Sie müssen hierbei nicht befürchten, dass die Dateien später alle im Hauptordner der Anwendung liegen. Das Setup kümmert sich in diesem Fall automatisch darum, dass gefundene Ordner angelegt und die Dateien entsprechend kopiert werden. Unsere Hilfedateien würden also auch wieder im Ordner "`docs`" liegen.
Allerdings kann es bei solchen sehr allgemeinen Angaben auch passieren, dass Dateien im Setup landen, die Sie eigentlich gar nicht weitergeben wollen. So würde zum Beispiel auch das Setupskript selbst berücksichtigt werden.

Nicht zu vergessen evtl. vorhandene temporäre Dateien. Zum Glück gibt es dafür einen weiteren Parameter, mit dem sich unerwünschte Dateien ausklammern lassen. Es sind mehrere Angaben möglich, die durch Komma voneinander zu trennen sind. Im folgenden Beispiel verwende ich die Angabe für das Setupskript (`*.iss`), sowie typische temporäre Dateien, die von Delphi angelegt werden

```
Source: "*"; Excludes: "*.iss, *.~*, *.dof, *.cfg, *.dcu"; \
  Flags: recursesubdirs; DestDir: "{app}"
```

Standardmäßig erlaubt der Inno Setup-Editor bzw. -Compiler keinen Zeilenumbruch. Das Beispiel wäre also eigentlich eine lange Zeile. Wenn Sie aber den Inno Setup Prä-Prozessor installiert haben, den ich ab Seite 40 genauer vorstellen werde, dann können Sie mit Hilfe des Backslash \ einen Zeilenumbruch setzen.

Meine Hoffnung ist, dass demnächst eine Version von Inno Setup veröffentlicht wird, die auch von sich aus Zeilenumbrüche akzeptiert, weil die gezeigte Zeile eben nur kompiliert werden könnte, wenn der ISPP vorhanden ist.

Obwohl die Benutzung von *Wildcards* die Arbeit vereinfacht, gibt es meiner Meinung nach zwei Nachteile

1. Fehlende aber evtl. wichtige Dateien werden durch *Wildcards* nicht erfasst, solange min. eine andere Datei vorhanden ist, die das Suchkriterium erfüllt. Nehmen wir an, Ihre Anwendung benutzt diverse Bibliotheken, die Sie aber nicht einzeln angeben, sondern mit dem Filter "`*.dll`" in das Setup integrieren. Dass dabei die Bibliothek "`Wichtig.dll`" fehlt, merken Sie möglicherweise nicht, da alle anderen Bibliotheken vom Filter erfasst werden und der Compiler daher auch nichts bemängelt. In so einem Fall wäre es zwar aufwändiger, aber auch besser, die Dateien einzeln anzugeben, damit Sie auf das Fehlen reagieren können.

2. Eine benutzerdefinierte Installation lässt sich etwas schwieriger realisieren. Damit ist ein Setup gemeint, bei dem der Anwender auswählen kann, welche Programmteile installiert werden sollen. Wenn Sie aber nur mit *Wildcards* arbeiten, gibt es natürlich auch keine echte Trennung der Komponenten. Sie sollten dann versuchen, trotz *Wildcards* eigene Trennungen durchzuführen. Das folgende Beispiel zeigt Ihnen die Trennung zwischen Anwendung, Quellcode und Dokumentation

```
[Files]
Source: "*.exe"; DestDir: "{app}";
Source: "source\*"; DestDir: "{app}\source"; Flags: recursesubdirs
Source: "docs\*"; DestDir: "{app}\docs"; Flags: recursesubdirs
```

   Mit dieser Variante hätten Sie sozusagen drei verschiedene Komponenten, und der Anwender kann selbst entscheiden, ob er auch den Quellcode und die Dokumentation auf seinen Rechner installieren möchte. Wie so eine benutzerdefinierte Installation aufgebaut wird, zeige ich später noch.

Für den Moment möchte ich Ihnen noch das Flag `isreadme` vorstellen, das besonders bei Textdateien (`*.txt`, `*.wri`, `*.doc`) sinnvoll ist. Damit erscheint auf der letzten Seite des Installationsprozesses eine zusätzliche Auswahlmöglichkeit, mit der der Anwender entscheiden kann, ob er die Textdatei nach der Installation lesen möchte

```
Source: "Liesmich.txt"; DestDir: "{app}"; Flags: isreadme
```

Dateien nicht komprimieren

Inno Setup ist so entwickelt worden, dass es immer eine einzige ausführbare Setupdatei erzeugt, die alle zu installierenden Dateien enthält. Am besten ist so ein Setup also mit einem selbstentpackenden Archiv vergleichbar. Wenn Sie einmal vor der Aufgabe stehen, mehrere hundert Megabytes installieren zu müssen, dann wäre Ihr Setup entsprechend riesig.

Nehmen wir an, Sie möchten eine CD mit einer Präsentation veröffentlichen. Diese beinhaltet diverse kleine Dateien, Word-Dokumente, Diagramme, Tabellen, usw. usw. Dazu kommen aber auch ein paar Videos und Audiodateien. Sie können nun die kleinen Dateien wie gehabt in das Setup aufnehmen, während Sie die Videos und Audiodateien unkomprimiert auf die CD brennen und im Setup nur für das spätere Kopieren auf das Zielsystem sorgen. Das funktioniert mit Hilfe des Flags `external` und der Konstanten "`{src}`". Diese Konstante bezeichnet den Ordner, in dem sich das Setup befindet. Nehmen wir also an, die Videos liegen in einem gleichnamigen Ordner unterhalb des Setups, und sie sollen auch in einen Ordner namens "`videos`" kopiert werden, dann würde die Anweisung so aussehen

```
[Files]
Source: "{src}\videos\*.avi"; DestDir: "{app}\videos"; \
  Flags: external
```

Auf diese Weise bleibt das Setup relativ klein, und trotzdem können die Videos von ihm kopiert werden. Sie können das Kopieren von solchen Dateien natürlich auch von weiteren Faktoren abhängig machen. Sollte Ihre Präsentation, bzw. Ihre Anwendung, auch ohne diese externen Dateien funktionieren, dann können Sie dem Anwender die Wahl überlassen, ob er sie kopieren will oder nicht. Wie das geht, zeige ich Ihnen im Kapitel über die Tasks ab Seite 43.

Schriftarten

Die Installation von Schriftarten ist an sich nur eine Erweiterung der normalen Dateiinstallation. Ich möchte es dennoch in einem eigenen Kapitel erläutern. Wir nehmen als Beispiel an, dass unsere Anwendung eine eigene Schriftart benutzt. Diese Schriftart soll global im System registriert werden. Damit wäre das Zielverzeichnis der Schriftartenordner von Windows, der in Inno Setup per "`{fonts}`"-Konstante erreichbar ist. Außerdem muss auch der Klarname der Schriftart angegeben werden, der nichts mit dem Dateinamen zu tun hat. Mit diesem Namen erscheint die Schriftart später in Auswahllisten, etwa von Word o.ä. Programmen

```
Source: "ffa.ttf"; DestDir: "{fonts}"; \
  FontInstall: "Futurama FontAlien"; \
  Flags: onlyifdoesntexist uninsneveruninstall
```

Die beiden benutzten Attribute sind bei der Installation von Schriftarten generell zu empfehlen. Mit `onlyifdoesntexist` wird sichergestellt, dass die Datei nur installiert wird, wenn sie noch nicht vorhanden ist. Das zweite Attribut `uninsneveruninstall` verhindert die Deinstallation der Schriftart beim Entfernen der Awendung.
Wie gesagt, denken Sie bitte daran, dass Ihre Schriftart mit der gezeigten Anweisung global im System registriert wird und damit auch von anderen Anwendungen benutzt werden kann. Wenn Sie die Schrift aber bei der Deinstallation ebenfalls entfernen lassen, dann ist das auch nicht weiter schlimm. In Dokumenten, in denen Sie sie verwendet haben, würde dann einfach nur wieder die Standardschrift erscheinen. Andere oder schlimmere Nebeneffekte hätte es nicht.

Zum Installieren von Schriftarten sind unter Windows 2000 und höher mindestens Hauptbenutzer-Rechte erforderlich.

Shell-Erweiterungen

Auch hinter einer Shell-Erweiterung steckt erst einmal nur eine ganz normale Dateiinstallation. Shell-Erweiterungen sind aber üblicherweise Bibliotheken, die zwei bestimmte Funktionen exportieren müssen, mit denen sie im System registriert und wieder entfernt werden können. Mit Hilfe des Flags `regserver` kann Inno Setup solche Bibliotheken nach dem Kopieren selbst registrieren und bei der Deinstallation wieder entfernen. Ein Beispiel könnte so aussehen

```
Source: "ShellExtensionLibrary.dll"; DestDir: "{sys}\ShellExt"; \
  Flags: regserver restartreplace uninsrestartdelete
```

Die beiden rot markierten Attribute sollten Sie, meiner Meinung nach, ebenfalls grundsätzlich bei Shell-Erweiterungen verwenden.

Nehmen wir an, dass Ihr Setup ein Update ist. Der Anwender besitzt noch eine installierte ältere Version mit der damals aktuellen Shell-Erweiterung. Diese ist aktiv und kann daher vom Update nicht ausgetauscht werden. Sie wissen ja sicher, dass man Dateien nicht löschen oder ersetzen kann, wenn diese gerade benutzt werden.
Mit Hilfe des Flags `restartreplace` kann das Setup automatisch einen Neustart anbieten, wenn es feststellt, dass die Shell-Erweiterung auf Grund einer schon vorhandenen Version nicht registriert werden kann. Ohne das Flag würden Sie nur eine Fehlermeldung sehen, was natürlich keinen besonders guten Eindruck macht.
Das gilt auch für den Deinstaller. Nicht verwendete Bibliotheken werden problemlos entfernt, aber in Benutzung befindliche Dateien können eben nicht gelöscht werden. In so einem Fall würde ebenfalls eine Fehlermeldung erscheinen, die die Anwender auffordert, bestimmte Dateien selbst zu löschen. Aus dem Grund sollten Sie auch `uninsrestartdelete` benutzen, mit dessen Hilfe das Setup einen Neustart anbietet und dafür sorgt, dass dabei die nicht mehr benötigten Shell-Erweiterungen gelöscht werden.

Der Compiler meckert ...

Wenn Sie den bisherigen Anweisungen gefolgt sind und das Skript versuchsweise kompilieren, dann wird der Compiler vermutlich das Fehlen einiger notwendiger Einträge bemängeln. Ergänzen Sie daher bitte die Sektion "`[Setup]`" mit den folgenden Zeilen

```
[Setup]
AppName=TestSetup
AppVerName=TestSetup 1.0
DefaultDirName={pf}\TestSetup
```

Hinweisen möchte ich erst einmal nur auf die Konstante "`{pf}`", die den "`Programme`"-Ordner des Systems repräsentiert. Gehen wir von einem typischen Windows-System aus, würde die Vorgabe für unser Setup dann also "`C:\Programme\TestSetup`" lauten. Diese Vorgabe kann natürlich vom Anwender nach Belieben geändert werden.

Wenn Sie das Skript jetzt kompilieren, sollten Sie eine Datei namens "`setup.exe`" im Unterordner "`Output`" finden. Sie können diese Voreinstellung natürlich auch ändern, und das Setup mit anderem Namen im gleichen Ordner wie das Skript erzeugen lassen. Ergänzen Sie dazu bitte die beiden Zeilen

```
[Setup]
OutputBaseFilename=TestSetup-1.0
OutputDir=.
```

Die Endung "`.exe`" wird vom Compiler automatisch ergänzt, und der Punkt steht für das relative aktuelle Verzeichnis. Achten Sie übrigens darauf, welchen Namen Sie für das Setup wählen. Mir ist

es einmal passiert, dass ich versehentlich den Namen der zu installierenden Anwendung gewählt habe, die dadurch natürlich überschrieben wurde. Im Zweifel wählen Sie also besser einen separaten Ordner für die Setupdatei, um keine vorhandenen Dateien zu überschreiben.

Kompressionsraten und -stufen

Standardmäßig verwendet der Compiler die *lzma*-Komprimierung, weil diese die besten Werte hat. Mit Hilfe des "Compression"-Parameters können Sie diese Vorgabe aber auch ändern, bzw. genauer einstellen.
Letzteres bedeutet, dass Sie zusätzlich auch angeben können, wie gut komprimiert werden soll. Der Standardwert ist in dem Fall `max`. Daneben gibt es aber auch noch `fast`, `normal` und `ultra`. Die Einstellung hängt aber sehr stark von Ihrem System und dem zur Verfügung stehenden Arbeitsspeicher ab. Unter Umständen kann es zu Problemen kommen, wenn Sie die höchste Stufe wählen

```
[Setup]
Compression=lzma/ultra
```

Als Alternativen stehen Ihnen noch die Methoden *zip* und *bzip* zur Verfügung, jeweils mit den Werten von 1 bis 9, die aber nicht ganz so gute Kompressionsraten erreichen wie *lzma*.

```
Compression=bzip/7
```

Eine weitere Möglichkeit, die Kompressionsrate zu beeinflussen, ist "`SolidCompression`". Ist dieser Wert "yes", komprimiert der Compiler alle Dateien am Stück. Das kann zu einer besseren Gesamtrate führen, wenn Sie bspw. sehr viele Textdateien haben, die sich ja allgemein sehr gut komprimieren lassen.

```
[Setup]
SolidCompression=yes
```

Der Nachteil ist, laut Inno Setup-Hilfe, dass sich die Performance verschlechtern kann. Und weil alle Dateien zudem in einem einzigen Stream gespeichert sind, fängt das Setup im Fehlerfall wieder von vorn an. Als negatives Beispiel wird ein Setup genannt, das auf mehrere Datenträger verteilt ist. Käme es hier zu einem Fehler beim Entpacken einer Datei, müsste der Anwender die erste Diskette wieder einlegen und alles noch einmal entpacken lassen, wollte er die Installation wiederholen.

Die Hilfe empfiehlt daher, diese Einstellung nicht zu verwenden, wenn das Setup sehr groß ist oder eben mehrere Datenträger benötigt.

Die benutzerdefinierte Installation

Jetzt wollen wir dem Anwender erlauben, einzelne Komponenten auszuwählen und damit nur bestimmte Programmteile zu installieren. Dazu benötigt das Setup zunächst eine neue Sektion namens "`[Components]`". Als Beispiel zeige ich Ihnen zwei verschiedene Komponenten

```
[Components]
Name: "main"; Description: "Programmdatei"; \
  Flags: fixed disablenouninstallwarning
Name: "source"; Description: "Quellcode"; \
  Flags: disableuninstallwarning
```

Sie sehen hier zuerst einen Namen. Dieser Name wird intern gebraucht und später den Dateien zugeordnet. Verwenden Sie daher bitte keine Leer- oder Sonderzeichen, sondern benutzen Sie einen möglichst einfachen, aber dennoch aussagekräftigen Begriff. Für die Beschreibung, die danach folgt, können Sie einen ausführlicheren Begriff benutzen, denn sie wird dem Anwender später gezeigt.

Das Flag `fixed` sorgt dafür, dass die Komponente nicht abgewählt werden kann. Es eignet sich daher nur für Komponenten, die in jedem Fall installiert werden sollen. In meinem Beispiel ist das eben die Anwendung selbst. Bei Ihnen könnten es auch gemeinsam genutzte Dateien sein, die zwingend erforderlich sind.

Das zweite Flag `disablenouninstallwarning` unterdrückt eine Meldung des Setups, die Sie sehen, wenn Sie versuchen, eine bereits installierte Komponente abzuwählen. Sie können nämlich das Setup mehrmals starten und evtl. fehlende Komponenten nachträglich installieren. Allerdings ist es zurzeit nicht möglich, Komponenten wieder zu entfernen. Und das erwartet man ja eigentlich, wenn man so ein Häkchen wieder entfernt. Stattdessen zeigt Ihnen das Setup einen entsprechenden Hinweis an, den Sie mit dem Flag unterdrücken können.

Jetzt können Sie noch Installationstypen definieren. Stellen Sie sich das als Gruppe vor. Jeder Typ definiert, welche Komponenten installiert werden sollen. Der Anwender muss also nicht erst umständlich die Komponenten auswählen, sondern er wählt zum Beispiel den für ihn passenden Typ, und die Komponenten werden dadurch automatisch ausgewählt. Typische Namen für Installationstypen sind bspw.

- vollständige Installation
- benutzerdefinierte Installation
- minimale Installation

Ob Sie solche Installationstypen verwenden, liegt bei Ihnen und hängt auch vom Umfang Ihres Setups ab. Bei sehr vielen möglichen Komponenten, sollten Sie mit Hilfe der Installationstypen eine sinnvolle Vorauswahl treffen, um dem Anwender das langwierige Aus- bzw. Abwählen einzelner Komponenten zu ersparen.
Für mein Beispiel möchte ich die oben erwähnten drei Typen erstellen

```
[Types]
Name: "full"; Description: "Vollständige Installation"
Name: "compact"; Description: "Minimale Installation"
Name: "custom"; Description: "Benutzerdefinierte Installation"; \
  Flags: iscustom
```

Die erste Angabe ist wieder ein interner Name, den wir für die Zuordnung der Komponenten benötigen. Danach folgt die Beschreibung des Typs, die in einer Auswahlliste über den Komponenten angezeigt wird.
Eine Besonderheit ist das Flag `iscustom` im dritten Typ. Es spielt keine Rolle, welcher Typ ausgewählt ist, sobald der Benutzer den Status irgendeiner Komponente irgendwie verändert, springt das Setup automatisch auf den Typ, der dieses Flag benutzt.

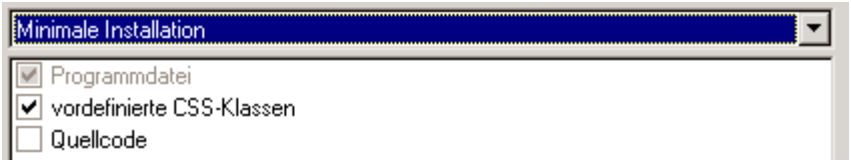
Allerdings fehlt noch die Zuordnung zu den Komponenten. Und dazu benötigen wir die internen Typennamen. Diese tragen Sie bei jeder Komponente ein, vorausgesetzt, die Komponente soll beim entsprechenden Typ vorausgewählt werden. Nehmen wir als Beispiel die Anwendung selbst, die immer installiert werden soll, und die auch gar nicht abgewählt werden kann. Für sie tragen wir alle drei Installationstypen ein

```
[Components]
Name: "main"; Description: "Programmdatei"; Flags: fixed; \
  Types: full compact custom
```

Im Gegensatz dazu soll der Quellcode nur bei der vollständigen Installation vorausgewählt sein

```
Name: "source"; Description: "Quellcode"; Types: full
```

Wählt der Anwender nun die minimale Installation, würde der Quellcode automatisch abgewählt werden, wie das folgende Bild eines anderen Setups zeigt



Denken Sie daran, dass diese Auswahl nichts mit der tatsächlichen Installation zu tun haben muss. Sie legen nur fest, dass eine bestimmte Komponente vorausgewählt wird, sobald ein dazu passender Installationstyp aktiv ist. Das letzte Wort hat aber immer noch der Anwender. Er wählt aus, was tatsächlich installiert wird.

Komponenten und Dateien verbinden

Damit fehlt nur noch die Verbindung zwischen den Komponenten und den zu installierenden Dateien. Momentan haben wir zwar die Liste und evtl. auch verschiedene Installationstypen, aber installiert wird nach wie vor immer noch alles. Um das zu ändern, ergänzen Sie die Angabe einer Datei um den Parameter "`Components`" und den internen Namen der gewünschten Komponente. Für die Hauptanwendung, die immer installiert wird, würde das so aussehen

```
[Files]
Source: "DelphiHTML.exe"; DestDir: "{app}"; Components: main
```

Für den Quellcode, der auf Wunsch installiert werden kann, würde die Zuordnung so aussehen

```
Source: "source\*"; DestDir: "{app}\source"; Flags: recursesubdirs \
  Components: source
```

Der "`Components`"-Parameter wird nicht nur bei Dateien verwendet. Auch Verknüpfungen, Registryeinträge usw. lassen sich so von der Auswahl bestimmter Komponenten abhängig machen.

Untergeordnete Komponenten

Sie können auch untergeordnete Komponenten erzeugen. Als Beispiel wollen wir davon ausgehen, dass Ihr Produkt zusätzliche Dokumentationen enthält. Damit ist nicht die normale Programmhilfe und das Handbuch gemeint, sondern vielleicht ergänzende Texte und ähnliches. Sie können das Setup nun so gestalten, dass eine Komponente namens "Dokumentation" anbietet. Diese Komponente enthält ihrerseits eigene Einträge für die normale und für die zusätzliche Dokumentation. So kann der Anwender entscheiden, ob er nur die normale oder aber auch die zusätzlichen Hilfen installieren möchte.

Dazu erzeugen Sie zunächst wie gewohnt die so genannte Elternkomponente

```
[Components]
Name: "docs"; Description: "Dokumentation"
```

Eine untergeordnete Komponente wird nun relativ einfach erzeugt, indem Sie einen weiteren Namen anhängen und per Backslash vom Namen der Hauptkomponente trennen. Und natürlich erhält jede untergeordnete Komponente ihre eigene Beschreibung

```
Name: "docs\appdoc"; Description: "Programmdokumentation"
Name: "docs\additional"; Description: "Ergänzende Dokumentationen"
```

Der Rest läuft wie bereits auf der vorigen Seite gezeigt. Dass der Name einer untergeordneten Komponente sozusagen zweiteilig ist, spielt keine Rolle. Er wird so übergegeben, wie Sie ihn hier definiert haben, zum Beispiel

```
[Files]
Source: "docs\*.hlp"; DestDir: "{app}\docs"; Components: docs\appdoc
Source: "docs\*.pdf"; DestDir: "{app}\docs"; \
  Components: docs\additional
```

Die Auswahlliste verbergen

Wenn Sie die Auswahlliste der Installationstypen komplett verbergen wollen, weil Sie sie nicht benötigen, dann definieren Sie bitte einen einzigen Typ mit dem Flag `iscustom`

```
[Types]
Name: EmptyList; Description: EmptyList; Flags: iscustom
```

Keine Ihrer Komponenten darf dann aber irgendeine Typzuordnung besitzen!

Verknüpfungen

Zu jedem Setup gehören in der Regel Verknüpfungen im Startmenü oder auf dem Desktop. Als Beispiel möchte ich eine Verknüpfung für das Programm im Startmenü unterbringen. Etwas verwirrend ist, dass die entsprechende Sektion "`[Icons]`" heißt. Zwar kommt das in etwa hin, denn eine Verknüpfung hat üblicherweise auch ein Symbol, aber dennoch sollte die Sektion in einer künftigen Version umbenannt werden.

Wie dem auch sei, bei einer Verknüpfung benötigen Sie den Namen der installierten Datei im Zielverzeichnis. Den Dateinamen selbst werden Sie als Setupentwickler natürlich kennen, und für das Zielverzeichnis gibt es wieder die "`{app}`"-Konstante.

```
[Icons]
Name: "{group}\TestSetup"; Filename: "{app}\MyProg.exe"; \
  WorkingDir: "{app}"
```

An erster Stelle sehen Sie hier eine neue Konstante namens "`{group}`". Sie verweist auf die Gruppe im Startmenü. Für diese Gruppe müssen Sie im Setup eine Vorauswahl treffen. Der Anwender hat aber die Möglichkeit, einen eigenen Namen zu verwenden. Die Vorauswahl treffen Sie in der "`[Setup]`"-Sektion

```
[Setup]
DefaultGroupName=TestSetup
```

Denken Sie beim Namen der Verknüpfung bitte daran, dass es sich um normale Dateien mit der Endung "`.lnk`" handelt. Es gelten also alle Bedingungen, die allgemein für Dateinamen gelten. Bestimmte Zeichen sind nicht erlaubt, und idealerweise sollte es nicht zwei Verknüpfungen mit dem gleichen Namen geben. Die möglichen Parameter in der Übersicht

Name	Erklärung
`Filename`	Name der zu öffnenden Datei
`Parameters`	Kommandozeilenparameter für das zu startende Programm
`WorkingDir*`	Arbeitsverzeichnis (s. Beispiel)
`Hotkey`	Hotkey für den Link
`Comment`	Kommentar (wird unter Windows 95/98/ME ignoriert)
`IconFilename`	Datei mit dem Symbol für die Verknüpfung
`IconIndex`	Index, wenn mehrere Symbole enthalten sind (null-basierend)
`Flags`	spezielle Einstellungen zum Programm (s. Hilfe)

Ich habe den Parameter "`WorkingDir`" aus gutem Grund mit einem Sternchen markiert. Es ist ein typischer Fehler, diesen Parameter wegzulassen. In einigen Fällen kommt es dann beim Starten einer Anwendung zu Fehlern. Dateien werden nicht gefunden, o.ä.
Die Ursache ist die, dass Ihr Programm erwartet, die notwendigen Dateien im aktuellen Ordner zu finden. Aber dieser aktuelle Ordner muss nicht mit dem Verzeichnis des Programms selbst identisch sein. Den Fehler können Sie mit geeigneten Funktionen beheben, die Sie in Ihrem Programm aufrufen. Trotzdem sollten Sie auf die Angabe des Arbeitsverzeichnisses nicht verzichten.

Für eine Desktopverknüpfung verwenden Sie die Konstante "`{userdesktop}`". Im Kapitel über die Mehrbenutzer-Umgebungen auf Seite 23 gehe ich auch noch auf die andere Variante ein, die für alle angemeldeten Benutzer gilt. Hier deshalb nur kurz eine Desktopverknüpfung für den aktuell angemeldeten Benutzer, der das Setup ausführt

```
[Icons]
Name: "{userdesktop}\TestSetup"; Filename: "{app}\MyProg.exe"; \
  WorkingDir: "{app}"
```

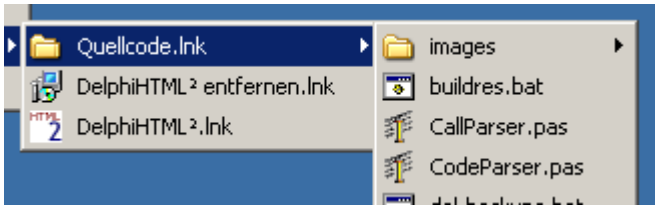
Shortcuts für Ordner

Ordnerverknüpfungen sind prinzipiell ähnlich. Sie lassen lediglich die Angabe des Arbeitsverzeichnisses weg. Als Beispiel legen wir eine Verknüpfung auf den Ordner mit den Quellcodes an, wenn diese installiert werden

```
Name: "{group}\Quellcode"; Filename: "{app}\sources"; \
  Components: sources
```

Verzeichnisverknüpfungen

Eine normale Ordnerverknüpfung habe ich Ihnen auf der vorigen Seite gezeigt. Dabei erscheint der Ordner als Eintrag im Startmenü. Wählen Sie den Eintrag aus, öffnet sich ein separates Explorerfenster mit dem Inhalt des Ordners.
Ab Windows 2000 gibt es spezielle Verzeichnisverknüpfungen. Auch hier wird der Name des Ordners angezeigt. Allerdings ist er diesmal ein Hauptmenüeintrag, und wenn Sie ihn auswählen, öffnet sich ein Untermenü mit allen gefundenen Dateien des Ordners, usw. Als Beispiel möchte ich Ihnen dieses Bild aus einem anderen Setup zeigen



Für eine solche Verknüpfung benötigen Sie die Version 5 von Inno Setup als Minimum. Und Ihr Betriebssystem, bzw. das Betriebssystem des Anwenders, der Ihr Produkt installiert, muss diese Verknüpfungen unterstützen. Sie sollten sie aber mit Bedacht einsetzen, denn sie macht nicht in jedem Fall Sinn.

Im Skript kommt nur das neue Flag `foldershortcut` hinzu, um aus einer normalen Verknüpfung für Ordner, wie auf der vorigen Seite gezeigt, eine solche spezielle Verknüpfung zu machen

```
Name: "{group}\Quellcode"; Filename: "{app}\sources"; \
  Components: sources; Flags: foldershortcut
```

Der Uninstaller im Startmenü: Ja oder Nein?

Normalerweise sollte man auf eine Verknüpfung für den Uninstaller im Startmenü verzichten. Es gibt so genannte Logo-Richtlinien von Microsoft, die auch den Aufbau von Setups beschreiben. Und nach diesen Richtlinien ist ausschließlich das Software-Modul in der Systemsteuerung für das Ändern oder Entfernen einer Anwendung zuständig.

Wenn Sie eine Verknüpfung für den Uninstaller anlegen möchten, dann verwenden Sie bitte die Konstante "`{uninstallexe}`"

```
Name: "{group}\Meine Anwendung entfernen"; Filename: "{uninstallexe}"
```

Sprachbarrieren

An dieser Stelle möchte ich auf die Sprache des Setups eingehen. Standardmäßig verwendet Inno Setup Englisch, was natürlich bei reinen deutschsprachigen Programmen unschön wirkt. Aber seit Version 4 enthält Inno Setup diverse Dateien, mit deren Hilfe sich die Sprache des Setups ändern lässt.
Diese ISL-Dateien finden Sie im Unterordner "`languages`" im Verzeichnis von Inno Setup. Zum Einbinden benötigen Sie ein frei wählbares Sprachpräfix und die gewünschte Datei. Der Compiler kann mit Hilfe der internen Konstanten "`compiler`" sein eigenes Verzeichnis finden, so dass Sie nur noch das gewünschte Unterverzeichnis angeben müssen. In dem Fall ist das der schon genannte "`languages`"-Ordner

```
[Languages]
Name: "de"; MessagesFile: "compiler:languages\German.isl"
```

Inno Setup unterstützt von Haus aus mehrsprachige Setups. Das bedeutet, Sie können mehr als eine Sprache angeben, und der Anwender kann dann vor der Installation die ihm genehme auswählen. Ergänzen Sie die gewünschten Sprachen so, wie eben gezeigt, bspw. Englisch

```
Name: "en"; MessagesFile: "compiler:Default.isl"
```

Bei einem mehrsprachigen Setup sollten Sie immer Englisch als erste Sprache angeben. Das hat mit der installierten Windowsversion zu tun. Die ISL-Dateien enthalten die Sprach-IDs, und das Setup kann erkennen, ob eine der IDs zum installierten Windows passt.
Bei einem deutsch/englischen Setup, das auf einem deutschen Windows gestartet wird, würde sich die Dateiauswahl daher auf Deutsch präsentieren und diese Sprache auch gleich als Vorauswahl anbieten.
Anders sieht es aus, wenn ich das folgende Beispiel auf einem deutschen Windows starte

```
[Languages]
Name: "nr"; MessagesFile: "compiler:languages\Norwegian.isl"
Name: "en"; MessagesFile: "compiler:Default.isl"
```

Keine der Sprach-IDs würde zur Windowsversion passen, und darum würde das Setup die erste Sprache benutzen, die es findet. Und das wäre Norwegisch. Wie gesagt, der Anwender kann natürlich aus der Liste auch Englisch auswählen, aber der Dialog selbst wird auch mit norwegischen Beschriftungen und Buttons usw. angezeigt.

Mehrsprachige Lizenztexte

Wenn Sie Lizenztexte benutzen, und diese ebenfalls abhängig von der Sprache anzeigen lassen möchten, dann benutzen Sie bitte den Parameter "`LicenseFile`" mit der jeweiligen Datei

```
[Languages]
Name: "de"; MessagesFile: "compiler:German.isl" \
  LicenseFile: "License-DE.rtf"
Name: "en"; MessagesFile: "compiler:Default.isl" \
  LicenseFile: "License-EN.rtf"
```

Mehrsprachigkeit richtig gemacht ...

Das Skript würde nun Englisch und Deutsch unterstützen, aber es gibt noch einen Nachteil: Ein paar eigene Meldungen sind nach wie vor nur deutsch. Das betrifft etwa die Namen der Komponenten und Installationstypen. In der Regel werden auch Namen von Verknüpfungen lokalisiert.
Mit Hilfe der Sektion "`[CustomMessages]`" kann das Problem gelöst werden. Nehmen wir als Beispiel die Angabe für den Uninstaller, die Sie erst einmal wie folgt deklarieren

```
[CustomMessages]
RemoveApp=Meine Anwendung entfernen

[Icons]
Name: "{group}\{cm:RemoveApp}"; FileName: "{uninstallexe}"
```

Mit Hilfe von "`{cm:}`" greifen Sie auf den Text der neuen Sektion zu. Am Ergebnis ändert das nichts. Die Verknüpfung heißt immer noch *Meine Anwendung entfernen*. Bei einem Setup mit mehreren Sprachen ergänzen Sie nun lediglich noch das in der Sektion "`[Languages]`" benutzte Sprachpräfix. Und Sie müssen eine Übersetzung für alle Sprachen schreiben, die Sie dort angegeben haben, bspw.

```
[CustomMessages]
de.RemoveApp=Meine Anwendung entfernen
en.RemoveApp=Remove My Application
```

Abhängig von der ausgewählten Sprache würde nun die Verknüpfung entweder auf Deutsch oder auf Englisch sein. Auf die gleiche Weise verfahren Sie bitte auch mit anderen Texten, zum Beispiel mit dem Namen der Anwendung

```
[Setup]
AppName={cm:MyAppName}
AppVerName={cm:MyAppName} 1.0

[CustomMessages]
de.MyAppName=Meine Anwendung
en.MyAppName=My Application
```

Weil für "`AppName`" nun eine Variable benutzt wird, deren Wert sich sprachabhängig ändert, entsteht ein kleines Problem mit einer anderen Direktive. Standardmäßig erzeugt der Compiler einen Versionsinfo-Block für Ihre Setupdatei, in dem u.a. der Wert von "`AppName`" als Beschreibung benutzt wird. Diese Beschreibung könnten Sie in den Dateieigenschaften Ihres Setups lesen.
Für diese Beschreibung sind aber Konstanten vorgesehen, deren Wert sich nicht ändert. Wie gesagt, es handelt sich um einen Infoblock in der EXE-Datei. Sie müssten also entweder auf die Beschreibung verzichten, oder Sie müssen sich für eine Sprache entscheiden und die Direktive von Hand deklarieren

```
[Setup]
VersionInfoDescription=My Application
```

*Verzeichnisnamen und Registryeinträge sollten Sie besser **nicht** lokalisieren! Hier ist es meist üblich, Englisch als Standard zu benutzen. Der Grund ist ganz einfach: Das Setup fragt Sie bei jedem Start nach der Sprache, auch wenn Ihre Anwendung bereits installiert ist. Wählen Sie nun also eine andere Sprache aus als die der schon installierten Version, dann werden neue Ordner und Registryeinträge usw. angelegt.*

Sprachabhängig kopieren

Die Mehrsprachigkeit des Setup lässt sich auch beim Kopieren verwenden. Das kann hilfreich sein, wenn das Setup verschiedene Sprachversionen Ihrer Anwendung enthält. Das können Ressourcendateien sein, die die Meldungen des Programms beeinflussen. Es können aber auch ganz einfach nur verschiedene Hilfedateien für jede Sprache sein. Ich zeige Ihnen alles an einem fiktiven Beispiel.
Wir nehmen an, dass wir ein Programm haben, das eine englischsprachige Oberfläche und englische Meldungen enthält. Dieses Programm muss immer installiert werden, egal welche Sprache der Anwender ausgewählt hat.

```
[Files]
Source: "MyProg.exe"; DestDir: "{app}"
```

Wenn jemand die deutsche Sprache für das Setup ausgewählt hat, dann soll eine zusätzliche Ressourcendatei installiert werden, die das Programm auf einem deutschen Windows übersetzen würde. Hierfür ergänzen wir den Parameter "`Languages`" mit dem von uns gewählten Sprachpräfix aus der "`[Languages]`"-Sektion

```
Source: "MyProg.de"; DestDir: "{app}"; Languages: de
```

Und zum Schluss sollen noch die Hilfedateien kopiert werden. Das Setup enthält sowohl die englischen als auch die deutschen Versionen. Welche kopiert werden, hängt wieder von der Auswahl des Anwenders ab

```
[Files]
Source: "help-DE\*.chm"; DestDir: "{app}\help"; Languages: de
Source: "help-EN\*.chm"; DestDir: "{app}\help"; Languages: en
```

Standardmeldungen des Installers austauschen

Wenn Sie die Standardmeldungen des Setups verändern möchten, dann werfen Sie bitte zuerst einen Blick in die entsprechende ISL-Datei. Dort finden Sie den Bezeichner für den Text, den Sie anpassen möchten. Aber ändern Sie nach Möglichkeit nicht die ISL-Datei, sondern erstellen Sie in Ihrem Setup eine eigene "`[Messages]`"-Sektion

```
[Messages]
de.ConfirmUninstall=Warum willst du denn %1 entfernen?
en.ConfirmUninstall=Why do you want to remove %1?
```

Denken Sie bei allen Angaben, ob in "`[Messages]`" oder "`[CustomMessages]`", immer an das Sprachpräfix. Lassen Sie es weg, dann gilt der Text global für alle Sprachen.

Die Registry

In diesem Kapitel möchte ich ein wenig auf die Registry eingehen. Speziell natürlich auf die Frage, wie man mit Inno Setup etwas in die Registry schreibt. Sie kennen sicher diese typischen Firmenschlüssel, etwa "`Software\Microsoft`", "`Software\Adobe`", usw.
Ob Sie einen solchen Schlüssel benötigen, hängt von Ihrer Anwendung ab. Es besteht keinerlei Zwang, einen solchen Schlüssel anlegen zu müssen. Üblicherweise werden Einstellungen zum Programm in diesen Schlüsseln gespeichert. Damit ergibt sich die nächste Frage: Handelt es sich um globale Einstellungen, oder soll jeder Anwender die Möglichkeit besitzen, sie zu ändern?

Sie sollten bedenken, dass auf NT-Betriebssystemen nicht jeder Benutzer Administratorrechte hat. Für die tägliche Arbeit ist das ohnehin nicht zu empfehlen. Globale Einstellungen, die für alle Benutzerkonten gelten, können deshalb nur im Schlüssel `HKEY_LOCAL_MACHINE` gespeichert werden. Ohne die passenden Rechte lassen sich solche Einstellungen aber nicht ändern.
Dagegen liegen benutzerabhängige Einstellungen im Schlüssel `HKEY_CURRENT_USER`. Diese können vom angemeldeten Benutzer verändert werden, und zudem kann jeder Benutzer seine eigenen persönlichen Einstellungen verwenden.

Unter Windows 95, 98 und ME spielen Benutzerrechte keine Rolle. Wenn Sie unter diesen Systemen entwickeln und Ihre Setups erzeugen, dann sollten Sie aber auch die Versionen Windows NT, 2000, XP und höher im Kopf behalten, um unschöne Fehler zu vermeiden, die in den meisten Fällen durch mangelnde Berechtigungen verursacht werden.

Als Beispiel möchte ich einen einfachen Schlüssel in der Registry anlegen, der global gültig sein soll. Demzufolge muss er nach `HKEY_LOCAL_MACHINE` (`HKLM`)

```
[Registry]
Root: HKLM; Subkey: "Software\Ihre Firma"; \
  Flags: uninsdeletekeyifempty
```

Dieser Schlüssel würde bei der Deinstallation nur entfernt werden, wenn er komplett leer ist. Das liegt an dem benutzten Flag und hat einen simplen Grund: Sie könnten mehrere Programme geschrieben haben, die sich alle innerhalb dieses Hauptschlüssels eintragen. Genau das ist auch der Grund, weshalb ich den Hauptschlüssel separat angebe.
Legen wir aber erst einmal den Anwendungsschlüssel an

```
Root: HKLM; Subkey: "Software\Ihre Firma\Ihre Anwendung"; \
  Flags: uninsdeletekey;
```

Und nun erkennen Sie auch den Sinn. Nehmen wir an, Anwender A besitzt nur ein einziges Programm von Ihnen. Deinstalliert er es, wird der o.g. Hauptschlüssel keine weiteren Angaben mehr enthalten und kann damit entfernt werden.
Anwender B hingegen hat noch ein anderes Programm von Ihnen installiert, das er auch behalten möchte. In seinem Fall würde der Hauptschlüssel also noch die Angaben dieses zweiten Programms enthalten. Würden Sie in Ihrem Setup den Hauptschlüssel nun ohne Rücksicht entfernen lassen, wäre der Schlüssel des zweiten Programms ebenfalls weg.

Der Anwendungsschlüssel unseres Programms wird auf Grund des Flags `uninsdeletekey` in jedem Fall entfernt, gleichgültig ob er noch Werte oder weitere Schlüssel enthält oder nicht.
Das Flag `uninsdeletekeyifempty` verhindert das jedoch und würde den Registryschlüssel nur entfernen, wenn dieser leer ist.

Damit fehlt eigentlich nur noch ein Wert. Für das Beispiel werde ich den Pfad der Anwendung eintragen

```
Root: HKLM; Subkey: "Software\Ihre Firma\Ihre Anwendung"; \
  ValueName: "Path"; ValueData: "{app}"; ValueType: string
```

Name	Erklärung
ValueName	Name des Eintrags
ValueData	Wert des Eintrags

ValueType	string	String-Wert (REG_SZ)
	expandsz	String-Wert (REG_EXPAND_SZ)
	multisz	String-Wert (REG_MULTI_SZ)
	dword	DWord
	binary	Binärwert

Die Sache mit den Admin-Rechten

Da ich die Einträge im Schlüssel HKLM anlege, würde das Setup nur funktionieren, wenn der Benutzer, der es ausführt, Hauptbenutzer- oder Administratorrechte hat. Melden Sie sich bitte einmal mit einem eingeschränkten Benutzerkonto an, und starten Sie das Setup. Wenn Ihr Windows richtig konfiguriert ist, dürften Sie die Einträge in der Registry nicht finden. Zudem sollte eine Fehlermeldung des Setups erscheinen, weil es die Registryeinträge nicht anlegen kann.

Ist dies nicht der Fall, und die Einträge werden angelegt, dann würde ich mir an Ihrer Stelle Gedanken machen. Sie sollten immer bedenken, dass das, was Sie tun können, auch ein Virus o.ä. Schadprogramm tun kann, das im gleichen Kontext läuft wie Ihr Benutzerkonto.

Sie sollten niemals auf die Idee kommen, und den Anwender auffordern, sein System in irgendeiner Form zu verstellen. Sie sollten eher sich selbst fragen, ob es zwingend nötig ist, auf spezielle Bereiche des Systems oder der Registry zuzugreifen. Ist es nicht unbedingt erforderlich, dann lassen Sie es.

Sollte es notwendig sein, dann müssen Sie dafür sorgen, dass Ihr Setup von einem Benutzer mit eingeschränkten Rechten gar nicht erst ausgeführt werden kann. Seit Version 5.1.9 sorgt Inno Setup selbst dafür, dass erstellte Setups mindestens Administratorrechte benötigen. Dieses neue Verhalten (u.a. eine Reaktion auf die Benutzerkontrolle von Windows Vista) können Sie mit Hilfe der Anweisung "PrivilegesRequired" in der "[Setup]"-Sektion an Ihre Bedürfnisse anpassen. In vielen Fällen dürften Hauptbenutzerrechte ausreichend sein

```
[Setup]
PrivilegesRequired=poweruser
```

Soll Ihr Setup auch von einem Benutzer mit eingeschränkten Rechten installiert werden können, dann verwenden Sie bitte none als Wert.

Dateitypen registrieren

Es gibt Anwendungen, die eigene Dateiformate benutzen. Das können Musikdateien sein, oder Dokumente jeder Art und dergleichen. Wenn der Entwickler möchte, dass man so ein eigenes Format durch direktes Anklicken im System benutzen kann, dann muss er dazu den Dateityp registrieren. Das bedeutet, man trägt die Dateiendung des eigenen Formats in einen bestimmten Bereich der Registry ein und ordnet ihr Aktionen zu, wie bspw. *Öffnen* oder *Drucken*.

Daraus leiten sich die allgemein anerkannten Regeln ab

1. Die Dateiendung wird als Schlüssel in `HKEY_CLASSES_ROOT` geschrieben und mit einem Verweis auf einen anderen internen Schlüssel, dem Typschlüssel, versehen.
2. Dieser interne Typschlüssel definiert eine Beschreibung und ein Symbol (optional), und er enthält die gewünschten Aktionen in Form von untergeordneten Schlüsseln.
3. Die Aktionen definieren Pfad und ggf. zusätzliche Parameter.

Für die erste Regel gibt es eine Ausnahme, auf die ich im Anschluss an die folgenden Erläuterungen eingehen werde. Beginnen wir jetzt erst einmal mit dem Hauptschlüssel, der gleichzeitig die gewünschte Dateiendung repräsentiert

```
[Registry]
Root: HKCR; Subkey: ".xyz"; ValueType: string; ValueName: ""; \
  ValueData: "XYZ.File"; Flags: uninsdeletevalue
```

Mit dieser Anweisung wird ein Registryschlüssel namens "`.xyz`" erzeugt, der für die Endung des eigenen Formats steht. Dieser Endung wird ein `string`-Wert zugeordnet, der in den so genannten Standardschlüssel geschrieben wird und auf den internen Typschlüssel verweist.
Der Standardschlüssel ist in üblichen Registryeditoren zwar mit "(Standard)" gekennzeichnet, aber das ist nicht sein tatsächlicher Name. Um ihn auszulesen oder ihm einen Wert zuzuweisen, benutzen Sie bitte einen leeren String für "`ValueName`".

Der interne Typschlüssel wird nun auf die gleiche Weise erzeugt. Wie Sie sehen, wird auch hier wieder der Standardwert mit einem `string` belegt. Dabei handelt es sich um die Beschreibung des Dateityps, die einigermaßen aussagekräftig sein sollte

```
Root: HKCR; Subkey: "XYZ.File"; ValueType: string; ValueName: ""; \
  ValueData: "Meine XYZ-Datei"; Flags: uninsdeletekey
```

Optional könnte man nun ein eigenes Symbol für den Dateityp festlegen. Dazu benötigt man einen speziellen untergeordneten Schlüssel, dessen Standardwert auf die Symboldatei und ggf. auch auf den Symbolindex verweisen muss. Bitte denken Sie daran, dass es sich hierbei nicht um eine Setupverknüpfung handelt. Es wird ein `string`-Wert in die Registry geschrieben, und wenn Sie in der benutzten Datei mehrere verschiedene Symbole haben, und wenn Sie deshalb einen Indexwert benutzen müssen, dann trennen Sie Dateinamen und Index durch Komma. Das folgende Beispiel zeigt, wie das zweite Symbol aus der Programmdatei benutzt wird. Da es sich bei den Symbolen um eine Null-basierende Zählweise handelt, ist der Wert für das zweite Symbol folgerichtig die 1

```
Root: HKCR; Subkey: "XYZ.File\DefaultIcon"; ValueType: string; \
  ValueName: ""; ValueData: "{app}\MyProg.exe,1"
```

Es fehlt die gewünschte Aktion. Darunter müssen Sie sich vorstellen, dass beim Anklicken einer Datei im Windows Explorer bspw. etwas passieren soll. Im einfachsten Fall startet Ihr Programm und zeigt den Inhalt der angeklickten Datei an.
Die Regel schreibt vor, dass so eine Aktion grundsätzlich im Schlüssel "`shell`" definiert wird, der seinerseits im internen Typschlüssel definiert ist. Der Name des Aktionsschlüssels erscheint im Kontextmenü der Datei, sofern nicht eine gesonderte Beschreibung angegeben wurde. Im Aktionsschlüssel muss dann noch zwingend der Schlüssel "`command`" definiert werden, dessen Standardwert die gewünschte Aktion auslösen soll. Für das Skriptbeispiel definieren wir also die Schlüsselfolge "`shell\open\command`". Das rot markierte Wort *open* ist übrigens nicht fest

vorgeschrieben. Es bietet sich in diesem Fall nur deshalb an, weil wir die Datei ja öffnen wollen. Eine Druckaktion wird überlicherweise mit dem *Verb print* gekennzeichnet. Aber Sie könnten ebenso auch die Namen von Bekannten und Verwandten verwenden

```
Root: HKCR; Subkey: "XYZ.File\shell\open\command"; \
  ValueType: string; ValueName: ""; \
  ValueData: """{app}\MyProg.exe"" ""%1"""
```

Zur Sicherheit sollten Sie die Pfadangaben und Parameter, die Dateien bezeichnen, in Anführungszeichen einschließen, um Probleme mit langen Pfad- und Dateinamen, bzw. mit Leerzeichen in selbigen, zu vermeiden.
Der Parameter "`%1`" ist nur ein Beispiel. Es würde bedeuten, dass Ihr Programm gestartet wird, und dass ihm die angeklickte Datei als erster Parameter übergeben wird. Welche Parameter in Ihrem Fall notwendig sind, hängt deshalb von Ihrem Programm ab.

Das Kontextmenü des Dateityps würde die neue Aktion bisher als *Open* anzeigen. Wenn Sie eine eigene Beschreibung möchten, schreiben Sie diese in den Standardwert des Aktions-schlüssels ein

```
Root: HKCR; Subkey: "XYZ.File\shell\open"; ValueType: string; \
  ValueName: ""; ValueData: "Mit meinem Programm öffnen"
```

Wenn Sie mehrere solcher Aktionen eintragen, dann können Sie auch eine Standardaktion festlegen, die bei einem Doppelklick auf die Datei ausgelöst wird. Dazu tragen Sie den internen Schlüssel der Aktion als Standardwert von "`shell`" ein

```
Root: HKCR; Subkey: "XYZ.File\shell"; ValueType: string; \
  ValueName: ""; ValueData: "open"
```

Nach dem Ausführen des Setup hätten wir damit bereits den neuen Dateityp. Allerdings zeigt er möglicherweise noch nicht das richtige Symbol an. Und darum lösen wir das Neuladen der internen Zuordnungen mit Hilfe einer speziellen Anweisung in der "`[Setup]`"-Sektion aus. Ein Neustart des Systems ist nicht erforderlich.

```
[Setup]
ChangesAssociations=yes
```

Ab Windows 2000 gibt es noch eine zweite Möglichkeit, Dateitypen lokal für den Benutzer zu registrieren. Es macht, meiner Ansicht nach, aber nicht viel Sinn, das in einem Setup zu benutzen, weil dann eben nur der gerade angemeldete Benutzer, der das Setup ausführt, einen Vorteil davon hätte.
Sorgen Sie stattdessen lieber dafür, dass bestimmte Rechte für Ihr Setup notwendig sind. Dann können Sie die Dateitypen global für alle Benutzerkonten registrieren.

INI-Dateien

Wenn Sie vor der Benutzung der Registry zurückschrecken oder keinen Grund sehen, Ihre Daten dort zu speichern, dann sind vielleicht INI-Dateien etwas für Sie. Allerdings möchte ich sagen, dass es mittlerweile kaum noch einen sinnvollen Grund gibt, per Setup auf eine INI-Datei zuzugreifen. Die Gründe liegen, meiner Meinung nach, auf der Hand

1. Ihr Programm sollte so entwickelt worden sein, dass es auf die INI-Datei zugreifen kann und sie ggf. auch erzeugt. Wenn Ihr Programm mit Benutzertrennung arbeitet und für jeden Benutzer eigene INI-Dateien anlegt, dann gibt es keinen Grund, eine solche Datei per Setup erzeugen zu wollen, weil die dann ja nur für den Benutzer angelegt werden würde, der das Setup ausführt. Und wenn Sie Ihre INI-Datei im Programmordner anlegen, handeln Sie sich unter Umständen Rechteprobleme ein.

2. Der Zugriff auf den Windows- oder andere Systemordner ist inzwischen völlig verpönt. Sie sollten Ihre eigenen INI-Dateien nicht dort ablegen, denn gerade in Systemordnern wird sehr viel Wert auf die Berechtigungen gelegt. Eingeschränkte Benutzerkonten haben zwar Lesezugriff auf die INI-Datei, aber eigene Werte schreiben können sie nicht.

Der Vollständigkeit halber möchte ich Ihnen aber trotzdem zwei Beispielzeilen zeigen, in denen der Pfad der Anwendung in einer INI-Datei gespeichert wird. Einmal im Programmordner, das zweite Mal im "`Anwendungsdaten`"-Ordner des aktuellen Benutzers. Das ist eine gute Gelegenheit, Ihnen die "`{userappdata}`"-Konstante zu zeigen, die auf eben diesen Ordner verweist

```
[Ini]
Filename: "{app}\TestIni.ini"; Section: "Setup"; \
  Key: "path"; String: "{app}"
FileName: "{userappdata}\MeinProgramm\Test.ini"; \
  Section: "Setup"; Key: "path"; String: "{app}"
```

Als letztes Beispiel in diesem Kapitel soll auch noch kurz gezeigt werden, wie man eine schon existierende INI-Datei erweitert und dafür sorgt, dass bei der Deinstallation nur der von uns neu eingefügte Eintrag entfernt wird. Nehmen wir also als Beispiel an, auch wenn es der o.g. Regel Nr. 2 widerspricht, dass Sie unbedingt einen Eintrag in der "`win.ini`" vornehmen müssen. Neu ist hier nur das Flag `uninsdeletesection`, mit dem der Uninstaller angewiesen wird, nur die neu erzeugte Sektion zu entfernen. Der Rest der "`win.ini`" wird nicht verändert

```
Filename: "{win}\win.ini"; Section: "NeueTestsektion"; \
  Key: "TestEintrag"; String: "Hallo, win.ini!"; \
  Flags: uninsdeletesection
```

Mehrbenutzer-Umgebungen

Ich habe die Benutzertrennung und die Berechtigungen ja bereits angesprochen. Das wirft die Frage auf: Wie sieht es bei Inno Setup generell mit Mehrbenutzer-Umgebungen aus? Sie haben sicher kein Interesse, ein Setup mehrfach starten zu müssen, nur um das zu installierende Programme unter verschiedenen Benutzerkonten verwenden zu können.
Üblicherweise installiert man das Programm einmal mit Administratorrechten, und es steht dann allen angemeldeten Benutzern zur Verfügung. Dabei werden eigentlich nur die Verknüpfungen so angelegt, dass alle Benutzer darauf zugreifen können.

Und das funktioniert auch mit Inno Setup, teilweise sogar automatisch.

Wenn Sie bspw. das Setup mit Administratorrechten starten, entspricht die schon erwähnte Konstante "`{group}`" mit dem Namen des anzulegenden Startmenüordners automatisch dem Startmenü aller Benutzer. Nur wenn Sie das Setup mit eingeschränkten Rechten ausführen, wird das Startmenü des aktuell angemeldeten Benutzers verwendet.

Dieses Verhalten können Sie mit der Direktive "`AlwaysUsePersonalGroup`" überschreiben. Das eignet sich bspw. für Programme, die direkt auf Systemfunktionen zugreifen, bzw. die neue Systemdateien installieren, die nur Administratoren zur Verfügung stehen sollen. Da die anderen Benutzerkonten mit solchen Programmen nichts anfangen können, ist es zweckmäßig, sie ihnen gar nicht erst zu zeigen. Wenn Sie dafür sorgen, dass das Setup nur von Administratoren gestartet werden kann, dann können Sie mit der genannten Direktive dafür sorgen, dass auch nur das Startmenü des Administrators für die Verknüpfungen benutzt wird

```
[Setup]
AlwaysUsePersonalGroup=yes
```

Im Kapitel über die Verknüpfungen auf Seite 13 habe ich es bereits angesprochen, dass es zwei Varianten gibt. Die Ihnen bekannte Konstante "`{userdesktop}`" gilt nur für den angemeldeten Benutzer, der auch das Setup gestartet hat. Die Konstante, mit der Sie eine Verknüpfung auf dem Desktop aller Benutzer anlegen, lautet dagegen "`{commondesktop}`".

Weitere Konstanten, von denen es eine Benutzer- und eine allgemeine Version gibt, sind

`userdesktop`	`commondesktop`	Desktopordner
`userappdata`	`commonappdata`	Anwendungsdaten
`userdoc`	`commondocs`	"Eigene Dateien"
`userfavorites`	`commonfavorites`	Favoriten
`userprograms`	`commonprogramms`	"Programme" im Startmenü
`userstartmenu`	`commonstartmenu`	Startmenü
`userstartup`	`commonstartup`	Autostart
`usertemplates`	`commontemplates`	Vorlagen

Aufräumen beim Installieren und Löschen

In diesem Kapitel möchte ich Ihnen zeigen, wie Sie während der Installation und während der Deinstallation Dateien löschen können. Viele Anwendungen erzeugen bestimmte Dateien erst beim Start. Das Setup weiß nichts von solchen Dateien und kann sie deshalb bei der Deinstallation auch nicht entfernen.
Als Beispiel möchte ich HLP-Hilfedateien nennen, die beim Aufruf versteckte GID-Dateien im gleichen Ordner anlegen. Eine Anweisung zum Löschen solcher Dateien würde daher dann so aussehen

```
[UninstallDelete]
Type: files; Name: "{app}\docs\*.gid"
```

Ich möchte und muss Sie bei solchen Löschvorgängen um absolute Sorgfalt und Vorsicht bitten. Es ist nämlich auch möglich, komplette Ordner zu löschen, ohne dass Sie die darin enthaltenen Dateien kennen müssen. Das folgende Beispiel würde den kompletten Ordner Ihres Programms entfernen und damit auch alle evtl. vorhandenen Konfig-, temporären und sonstigen Dateien löschen

```
Type: filesandordirs; Name: "{app}"
```

Das Problem ist nur, dass Ihnen niemand garantieren kann, dass der Anwender Ihr Programm in einen eigenen Ordner installiert hat. Er hat die freie Wahl. Er kann ein Verzeichnis nach eigenem Gutdünken auswählen. Und das kann auch ein schon vorhandenes Verzeichnis mit schon existierenden Dateien sein.
Bei einer normalen Deinstallation gibt es dabei keine Schwierigkeiten, weil das Setup jede Datei vermerkt, die es installiert. Ebenso jede Verknüpfung, jeden Registryeintrag, usw. Wenn Sie die Anwendung deinstallieren, dann werden nur die installierten Dateien und sonstigen Einträge entfernt. So weit, so gut. Wenn Sie nun aber eine Löschanweisung haben, die den kompletten Ordner löscht, dann werden dadurch natürlich auch alle anderen Dateien entfernt, die sich noch in dem Ordner befinden.

Deswegen lautet mein unbedingter Rat an Sie: Auch wenn es aufwändiger ist, alle zu löschenden Dateien einzeln anzugeben, tun Sie es! Sie ersparen sich und dem Anwender unter Umständen viel Ärger.

Noch eine kleine Besonderheit soll erwähnt werden: Die Sektion "`[UninstallDelete]`" wird schon während der Installation bearbeitet, weil die Einträge in die Log-Datei des Uninstaller geschrieben werden.
Wenn Sie Prüffunktionen und Abfragen benutzen, etwa die Frage an den Anwender, ob die Datei wirklich gelöscht werden soll, dann sehen Sie selbige bereits beim Setup, was evtl. etwas verwirrend ist. Verzichten Sie also nach Möglichkeit auf solche Dinge.

Das Gegenstück zu "`[UninstallDelete]`" ist "`[InstallDelete]`". Der Aufbau der Sektion entspricht "`[UninstallDelete]`", darum spare ich mir ein Beispiel. Benutzen Sie die Sektion, wenn Ihre Anwendung zum Beispiel ein Update ist, und wenn Sie bereits existierende Dateien einer älteren installierten Version löschen möchten, die in der aktuellen Version nicht mehr benötigt werden.

Programme starten

In eine ähnliche Kategorie wie das Löschen von Dateien und Ordnern fällt auch der Start von externen Programmen. Sie kennen das sicher aus einigen Setups, die Ihnen nach der Installation anbieten, die Anwendung gleich zu starten. Einige andere Setups tun das von selbst, weil die externen Programme wichtig für irgendwelche Einstellungen sind, usw.
Inno Setup erlaubt es Ihnen, Programme bei der Installation und bei der Deinstallation zu starten. Die dafür erforderlichen Sektionen heißen "`[Run]`" und "`[UninstallRun]`" und sind im Aufbau identisch. Darum möchte ich Ihnen nur das bereits erwähnte Beispiel zeigen, dass die installierte Anwendung auf Wunsch nach der Installation startet

```
[Run]
Filename: "{app}\MyProg.exe"; \
  Description: "Meine Anwendung jetzt starten"; \
  WorkingDir: "{app}"; \
  Flags: postinstall nowait skipifsilent unchecked
```

An erster Stelle sehen Sie den Dateinamen, gefolgt von einer Beschreibung und der Angabe des Arbeitsverzeichnisses. Die Erklärungen zu den benutzten Flags habe ich in der folgenden Tabelle zusammengefasst

Name	Erklärung
`postinstall`	Das Setup erzeugt eine Checkbox, mit deren Hilfe der Anwender selbst entscheiden kann, ob er das Programm starten will oder nicht. In dem Fall wird die benutzte Beschreibung angezeigt.
`nowait`	Das Setup wartet nicht auf die Ausführung des Programms.
`skipifsilent`	Läuft das Setup im so genannten Silent-Modus (sprich: der Installation ohne weitere Benutzerabfragen), dann wird der Start des Programms übergangen.
`unchecked`	Das Programm wird standardmäßig nicht gestartet. Es liegt beim Anwender, diese Aktion zuzulassen.

Weitere Möglichkeiten für die "[Setup]"-Sektion

Als kleine Zusammenfassung möchte ich hier eine Übersicht über weitere interessante Möglichkeiten geben, die Sie für Ihre Installationen nutzen können.

Name	Erklärung
`DiskSpanning (yes/no)`	Damit kann die Installation in diskettengerechte "Häppchen" aufgeteilt werden.
`DiskSliceSize (1457664)`	Legt die maximale Größe für die einzelnen Disketten-Teile fest. Wird ignoriert, wenn "`DiskSpanning`" nicht benutzt wird.
`DiskClusterSize (512)`	Clustergröße des Zielmediums (512 = Diskette). Wird ignoriert, wenn "`DiskSpanning`" nicht benutzt wird.

`AllowNoIcons (yes/no)`	Erlaubt dem Benutzer, das Anlegen von Programmgruppen mit Hilfe eine Checkbox zu verhindern
`AppMutex (string)`	Verhindert, dass eine Installation ausgeführt wird, solange eine Anwendung noch läuft. Programmierer werden den Begriff "Mutex" kennen. Er wird in vielen Fällen verwendet, um nur eine Instanz eines Programms starten zu können. Inno Setup kann auf diese Weise herausfinden, ob das Programm läuft und verweigert dann die Installation, weil normalerweise Dateien, die gerade benutzt werden, nicht überschrieben werden können.
`DisableStartupPrompt (yes/no)`	Verhindert die manchmal doch recht nervige Dialogbox "*xy* wird jetzt installiert" ☺
`MinVersion (9x,NT)`	Definiert die Betriebssystemversion, die für die Installation zulässig ist. Die Angabe von "`MinVersion=0,4`" erlaubt bspw. nur die Installation unter NT 4 und höher. Unter Windows 9x erscheint dann eine Fehlermeldung.
`Uninstallable (yes/no)`	Deaktiviert den Uninstaller (no). Eignet sich speziell für Systemupdates o.ä., die nicht entfernt werden sollen oder dürfen.
`DirExistsWarning`	Schaltet die Warnung ab, wenn ein existierendes Verzeichnis als Ziel gewählt wird.

`WindowVisible (yes/no)`	Benutzt den Vollbildmodus, so wie es früher bei Setups üblich war.

`WizardImageFile`	164x314 große Bitmap, die auf der Begrüßungs- und auf der Abschlussseite angezeigt werden soll.
`WizardSmallImageFile`	55x55 große Bitmap für den Headerbereich des Setup.
`SetupIconFile`	Ändert das Symbol der Setup.exe; das funktioniert aber nur unter NT-Betriebssystemen.

Codespielereien

Ich möchte jetzt auf die "[Code]"-Sektion zu sprechen kommen. Seit Version 4 enthält Inno Setup eine eingebaute Pascal-Skriptsprache. Damit lassen sich eine ganze Reihe von Dingen anstellen, die mit dem Skript so nicht möglich waren und sind.
Grundsätzlich sollten Sie ein wenig Erfahrung mit der Sprache Delphi haben. Wenn Sie keine Erfahrung haben, dann schauen Sie sich am besten ein paar Grundlagenbeiträge zu Delphi an. Bedenken Sie aber, dass die Skriptsprache von Inno Setup nicht die Möglichkeiten bietet, die Delphi hat. Dennoch würde es den Rahmen dieser Dokumentation sprengen, wollte ich hier solche Grundlagen auch noch besprechen.

Einen Link zur Homepage anzeigen

Zur Einstimmung beginnen wir mit einer relativ einfachen Geschichte. Wir wollen auf dem Programm einen zusätzlichen Link zur Homepage unterbringen. Um gleich noch einen anderen Nutzen daraus zu ziehen, gehen wir davon aus, dass wir immer noch ein zweisprachiges Setup (Englisch und Deutsch) besitzen. Darum benötigen wir erst einmal zwei Texte in der schon erwähnten Sektion "[CustomMessages]" (vgl. Seite 15)

```
[CustomMessages]
de.HomePageLink=Homepage besuchen
en.HomePageLink=Visit homepage
```

Der Link soll durch ein so genanntes Label vom Typ TNewStaticText dargestellt werden. Wir erzeugen es am besten in der Funktion "InitializeWizard". Dabei handelt es sich um eine von Inno Setup zur Verfügung gestellte Ereignisfunktion, die immer aufgerufen wird wenn die Oberfläche geladen wird.

```
[Code]
procedure InitializeWizard;
```

Wir beginnen mit zwei Variablen. Eine davon ist das Label, das wir erzeugen wollen, die andere ist der Abbrechen-Button des Assistenten

```
var
  UrlLabel  : TNewStaticText;
  CancelBtn : TButton;
begin
  CancelBtn := WizardForm.CancelButton;
```

der als Referenz für die Position unseres Labels dienen soll. Das Label selbst wird durch die Methode "Create" des TNewStaticText-Objektes erzeugt

```
  UrlLabel      := TNewStaticText.Create(WizardForm);
  UrlLabel.Top  := CancelBtn.Top + (CancelBtn.Height div 2) -
    (UrlLabel.Height div 2);
  UrlLabel.Left := WizardForm.ClientWidth - CancelBtn.Left -
    CancelBtn.Width;
```

Jetzt benötigen wir noch den Text für unser Label. Da sich der in "[CustomMessages]" befindet, brauchen wir die Funktion "ExpandConstant", mit der wir auf diverse Skriptwerte zugreifen können

```
  UrlLabel.Caption    := ExpandConstant('{cm:HomePageLink}');
```

Damit es auch wie ein Hyperlink aussieht, unterstreichen wir den Text, und wir benutzen den Hand-Cursor für das Label

```
  UrlLabel.Font.Style := UrlLabel.Font.Style + [fsUnderline];
  UrlLabel.Cursor     := crHand;
```

Als Farbe könnte man nun Blau festlegen, wovon ich aber abraten möchte. Jeder Benutzer kann sein eigenes Farbschema einstellen. Und es wäre denkbar, dass jemand Blau als Hintergrund für Fenster benutzt. Der Link wäre dann nicht mehr zu sehen. Zum Glück unterstützt Inno Setup bestimmte Farbwerte, die Sie vielleicht schon von Delphi kennen.
Der für uns interessante Wert heißt *clHighlight* und enthält die im System eingestellt Farbe für die Auswahlmarkierung in Menüs oder Listenelementen.

```
  UrlLabel.Font.Color := clHighlight;
```

Damit beim Anklicken unseres Links auch tatsächlich etwas passiert, wird eine so genannte "`OnClick`"-Prozedur benötigt

```
  UrlLabel.OnClick    := @UrlLabelClick;
```

Und zu guter Letzt muss das Label erfahren, wohin es gehört

```
  UrlLabel.Parent     := WizardForm;
end;
```

Das Label wäre damit bereits sichtbar, allerdings würde der Compiler die fehlende Klickprozedur bemängeln. Die fügen Sie im Code bitte vor der "`InitializeWizard`"-Prozedur ein

```
const
  Url = 'www.IhreHomepage.de';

procedure UrlLabelClick(Sender: TObject);
var
  errorCode : integer;
begin
  ShellExec('open',Url,'','',SW_SHOWNORMAL,ewNoWait,errorCode);
end;
```

Voilà. Wenn Sie das Skript kompilieren und das Setup starten, dann sollten Sie das neue Label eigentlich sehen.
Und wenn Sie es anklicken, müsste sich Ihr Browser öffnen und die angegebene Seite laden.

Im Inno Setup-Beispiel "`CodeClasses.iss`" wird nach diesem Muster übrigens ein Hilfe-Button erzeugt, den Sie zur Anzeige von zusätzlichen Informationen nutzen könnten.

Der Check-Parameter

Mit dem "Check"-Parameter können Sie die Installation einer Datei oder das Erzeugen einer Verknüpfung o.ä. von bestimmten Bedingungen abhängig machen. Nehmen wir als Beispiel an, Sie wollen ein bereits installiertes Programm zur Anzeige Ihrer Dateien verwenden. Allerdings soll das Setup auch ausgeführt werden, wenn dieses externe Programm nicht vorhanden ist. Eine Prüfung und ein Abbruch kommen deshalb nicht in Frage. Sie wissen aber, dass das Programm über einen Eintrag in der Registry zu finden ist.

Der "Check"-Parameter ermöglicht es Ihnen nun, eine bool-Funktion zu schreiben, in der Sie die gewünschte Bedingung überprüfen und je nach Erfolg TRUE oder FALSE zurückgeben. Abhängig davon wird die Datei kopiert oder nicht, bzw. die Verknüpfung o.ä. wird angelegt oder nicht.
Bei dem angesprochenen Beispiel soll es also darum gehen, den Eintrag des gesuchten Programms aus der Registry zu ermitteln und für die Verknüpfung zu verwenden, was wie folgt aussehen kann

```
[Icons]
Name: "{group}\Verknüpfung"; \
  Filename: "{reg:HKLM\Software\...\AppPaths,Path|}\Viewer.exe";
  Parameters: "{app}\MeineDatei.bin"
```

Mit der rot markierten Anweisung wird der Pfad eines Programms aus der Registry gelesen und mit dem Namen des Programms ergänzt. Die Verknüpfung würde also das Programm starten und die installierte Datei anzeigen, wenn das Programm vorhanden ist. Und das können Sie mit Hilfe von "Check" überprüfen

```
  Check: DoesTheViewerExist
```

Er verweist auf eine Funktion, in der wir prüfen, ob es das Programm überhaupt gibt

```
[Code]
function DoesTheViewerExist: boolean;
var
  tmp : string;
begin
  // den Wert aus der Registry holen, ...
  Result := (RegQueryStringValue(HKEY_LOCAL_MACHINE,
    'Software\...\AppPaths','Path',tmp)) and

  // ... & prüfen ob die Datei vorhanden ist
    (FileExists(tmp + '\Viewer.exe'));
end;
```

Das Prinzip ist recht einfach, wie Sie sehen. Sie lesen zunächst den Pfad aus der Registry aus. Dabei handelt es sich um den gleichen Pfad, der auch oben (rot markiert) in der Verknüpfung benutzt wurde. Dann prüfen Sie mit Hilfe von "FileExists", ob die gesuchte Datei existiert.

Der Rückgabewert der Funktion setzt sich in diesem Fall aus der Dateiprüfung und dem Registryzugriff zusammen. Sollte der Wert nicht ausgelesen können, weil er nicht existiert, dann ist die Funktion hier bereits FALSE, und die Verknüpfung würde nicht angelegt werden. Das gleiche gilt für die Dateiprüfung. Wird die Datei nicht gefunden, ist dieser Teil der Rückgabe FALSE, und bei einer and-Verknüpfung wäre dann das ganze Ergebnis FALSE.

Die Verknüpfung wird also nur angelegt, wenn der Registrywert ausgelesen werden konnte, und wenn im ermittelten Pfad die gesuchte Datei existiert.

Ein eigenes Optionsfenster erzeugen

Mit dem Optionsfenster möchte ich das Kapitel eigener Seiten im Setup beginnen. Wir stellen uns dazu folgende Situation vor: Ihre Anwendung benötigt einen bereits installierten externen Betrachter auf dem Zielsystem des Anwenders. Den kann es allerdings in verschiedenen Versionen geben, weil das System die Betrachter nicht überschreibt, sondern parallel installiert.
Und deshalb wollen Sie dem Anwender die Wahl überlassen, welchen Betrachter er verwenden möchte, wenn er denn schon verschiedene Versionen installiert hat.
Wir gehen davon aus, dass wir zwei verschiedene Versionen des Betrachters gefunden haben, die wir dem Anwender zeigen wollen.

Um eine Optionsseite zu erzeugen, benötigen wir eine globale `TInputOptionWizardPage`-Variable. Global deshalb, weil wir in zwei verschiedenen Prozeduren auf sie zugreifen müssen.
Die Seite selbst wird mit dem Befehl "`CreateInputOptionPage`" erzeugt. Der erste Parameter definiert die Seite, nach der der Optionsdialog angezeigt werden soll. Ich habe mich dafür entschieden, sie nach der Auswahl der Programmgruppe zu zeigen. Der entsprechende Wert heißt also *wpSelectProgramGroup*.
Die nächsten drei Parameter definieren den Titel, eine erläuternde Beschreibung für den Titel und eine allgemeine Beschreibung der Dialogseite. Diese Texte habe ich in meinem Fall wieder in der Sektion "`[CustomMessages]`" abgelegt.
Der vierte Parameter ist vom Typ `bool` und entscheidet darüber, ob die Dialogseite Checkboxen oder Radiobuttons benutzt. Der fünfte und letzte Parameter ist nützlich, wenn Sie sehr viele Optionen anbieten wollen, die unter Umständen den Rahmen des Fensters sprengen würden. In dem Fall ist es möglich, die Optionen innerhalb einer Listbox anzuzeigen.

Und so sieht der Befehl dann aus

```
procedure InitializeWizard;
begin
  DExplorePage := CreateInputOptionPage(wpSelectProgramGroup,
    ExpandConstant('{cm:DocSet}'),
    ExpandConstant('{cm:DExploreHead}'),
    ExpandConstant('{cm:DExploreDesc}'),
    true,
    false);
```

Die Seite würde damit bereits erzeugt und angezeigt werden. Aber sie enthält noch nichts weiter, was zu Fehlern führt. Darum ergänzen wir noch die beiden Betrachter. Ich verwende für das Beispiel zwei `strings` mit den Dateinamen

```
  DexplorePage.Add('c:\Betrachter1\Betrachter1.exe');
  DexplorePage.Add('c:\Betrachter2\Betrachter2.exe');
```

und ich stelle den ersten Betrachter als vorausgewählten Standard ein

```
  DExplorePage.Values[0] := true;
end;
```

Und jetzt müssen wir dafür sorgen, dass die Auswahl des Anwenders auch berücksichtigt wird. Dazu benötigen wir eine andere Ereignisfunktion, wobei Sie die Wahl haben. Sie können bspw. "`NextButtonClick`" benutzen, oder Sie verwenden "`CurStepChanged`". Es funktioniert beides, wie ich Ihnen zeigen werde.

Die Funktion "`NextButtonClick`" wird immer ausgelöst, wenn der Anwender auf den Weiter-Button klickt. Als Parameter wird dieser Funktion immer die ID der aktuellen Seite übergeben. Sie prüfen also lediglich, ob Ihre neue Seite erreicht ist. Die `TInputOptionWizardPage`-Variable (eigentlich handelt es sich um eine Klasse) stellt dafür die Eigenschaft `ID` zur Verfügung.

```
function NextButtonClick(CurPageID: integer): boolean;
begin
  if(CurPageID = DExplorePage.ID) then
  begin
```

Nun fragen wir ab, welche Option ausgewählt worden ist. Die Zählweise ist Null-basierend, das heißt, die erste Option (= der erste Betrachter) hat den Wert Null, der zweite Eins, usw. Das lässt sich am einfachsten in einer `case`-Anweisung realisieren

```
    case DExplorePage.SelectedValueIndex of
      1  : SelectedViewerFile := 'c:\Betrachter2\Betrachter2.exe';
      else SelectedViewerFile := 'c:\Betrachter1\Betrachter1.exe';
    end;
  end;
```

"`SelectedViewerFile`" ist eine `string`-Variable, die den Pfad und Namen des ausgewählten Betrachters speichert. Diese Variable benötigen wir später für die Verknüpfung.
Für den reibungslosen Ablauf des Setup setzen wir das Rückgabeergebnis der Funktion immer auf `TRUE`

```
  Result := true;
end;
```

Bevor ich zur Verknüpfung komme, möchte ich noch die zweite, schon angesprochene Variante mit "`CurStepChanged`" zeigen. Hier handelt es sich um eine Prozedur, die bei bestimmten Ereignissen aufgerufen wird. In unserem Beispiel würde uns das Ereignis kurz vor der Installation interessieren, denn wir wollen ja den Pfad des ausgewählten Betrachters wissen, bevor die Verknüpfung angelegt wird. Der Parameter dieser Prozedur bezeichnet den aktuellen Installationsschritt. Wir benötigen `ssInstall`. Und von da an unterscheidet sich der Code nicht mehr von der bereits gezeigten "`NextButtonClick`"-Version

```
procedure CurStepChanged(CurrentStep: TSetupStep);
begin
  if CurrentStep = ssInstall then
  begin
    case DExplorePage.SelectedValueIndex of
      1  : SelectedViewerFile := 'c:\Betrachter2\Betrachter2.exe';
      else SelectedViewerFile := 'c:\Betrachter1\Betrachter1.exe';
    end;
  end;
end;
```

Nachdem der ausgewählte Betrachter nun in der `string`-Variablen gespeichert wurde, können Sie ihn in der Verknüpfung nutzen. Da Sie, meines Wissens nach, nicht direkt auf die Variable zugreifen können, benötigen Sie eine Hilfsfunktion, die den Betrachter als Ergebnis liefert

```
[Icons]
Name: "{group}\{cm:DocSet}"; FileName: "{code:GetViewer}"; ...

[Code]
function GetViewer(const dummy: string): string;
begin
  Result := SelectedViewerFile;
end;
```

Eine zusätzliche Verzeichnisauswahl

Der zweite neue Dialog stellt Ihnen eine zusätzliche Verzeichnisauswahl zur Verfügung und ist an sich nur eine Variation des gerade gezeigten Prinzips. Eine zusätzliche Verzeichnisauswahl kann nützlich sein, wenn Sie einen bestimmten Ordner als Arbeits- oder sonstiges Verzeichnis für Ihr Programm festlegen wollen.
Dazu benötigen Sie zuerst eine Variable vom Typ `TInputDirWizardPage` und einen `string` für den ausgewählten Ordner

```
[Code]
var
 UserDirPage : TInputDirWizardPage;
 UserFolder  : string;

function InitializeSetup: boolean;
begin
 UserFolder  := '';
 Result      := true;
end;
```

Der Befehl zum Erzeugen der Verzeichnisseite lautet "`CreateInputDirPage`". Der erste Parameter ist wieder die ID der Seite, nach der die Verzeichnisauswahl angezeigt werden soll. Für das Beispiel habe ich mich dafür entschieden, die Auswahl direkt nach der Auswahl des Programmordners anzeigen zu lassen. Der Wert lautet also `wpSelectDir`.
Danach folgen drei Parameter, mit denen Sie Titel, die kurze Erläuterung des Titels und die etwas ausführlichere Beschreibung der Seite festlegen. Auch hier empfiehlt es sich, die Texte in der "`[CustomMessages]`"-Sektion abzulegen.
Der fünfte Parameter ist eine `bool`-Variable, die darüber entscheidet, ob der Vorgabewert an die Eingabe des Anwenders angehangen wird oder nicht. Der Vorgabewert ist dann der nächste und letzte Parameter

```
procedure InitializeWizard;
begin
  UserDirPage := CreateInputDirPage(wpSelectDir,
    ExpandConstant('{cm:PersonalDataLocation}'),
    ExpandConstant('{cm:PersonalDataHeader}'),
    ExpandConstant('{cm:PersonalDataDescription}'),
    false,
    '');
```

In diesem Beispiel habe ich keinen Vorgabewert benutzt und die `bool`-Variable auf `FALSE` gesetzt. So wird die Eingabe bzw. Auswahl des Anwenders als solche akzeptiert und verwendet. Alternativ dazu könnten Sie auch festlegen, dass immer ein festgelegter Ordnername angehängt wird. Mit anderen Worten: Egal, welchen Ordner der Anwender eingibt oder auswählt, der Vorgabewert wird immer angehängt. Dazu müssen Sie natürlich einen Vorgabewert verwenden, und Sie müssen die `bool`-Variable auf `TRUE` setzen

```
    ...
    true
    'New Folder');
```

Wenn Sie einen Ordner nicht nur auswählen lassen wollen, sondern wenn Sie möchten, dass der Anwender ggf. auch einen Ordner erstellen kann, dann verwenden Sie einen beliebigen Vorgabewert, setzen Sie aber die `bool`-Variable auf `FALSE`

```
    ...
    false
    'dummy');
```

Wenn der Anwender nun auf den Button durchsuchen klickt, erscheint im angezeigten Dialog zusätzlich ein Button, mit dem man einen neuen Ordner erzeugen kann.

Doch zurück zum Thema. Wie gesagt, die 3 Texte für den Titel und die beiden Beschreibungen sollten Sie wieder in der Sektion "`[CustomMessages]`" ablegen. Als Zusatz soll die Dialogseite aber auch die von der Standardverzeichnisauswahl bekannte Meldung "Klicken Sie Weiter ..." anzeigen. Sie könnten diese Meldung nun ebenfalls selbst deklarieren. Sie können aber auch direkt auf das Original von Inno Setup zugreifen.
Wenn Sie dazu einmal einen Blick in eine der ISL-Sprachdateien werfen, finden Sie dort die Angabe "`SelectDirBrowseLabel`" mit dem gesuchten Text. An diesen Text kommen wir mit der Funktion "`SetupMessage`" heran, wobei aber das Präfix **`msg`** vor den Bezeichner gesetzt werden muss

```
  UserDirPage.Add(SetupMessage(msgSelectDirBrowseLabel));
```

Beachten Sie bitte, dass es sich hier um keine `string`-Variable handelt. Es ist mehr mit einer internen Konstanten des Setups vergleichbar.

Als Voreinstellung soll nun noch der "`Anwendungsdaten`"-Ordner des aktuell angemeldeten Benutzers gesetzt werden. Zusätzlich sollten Sie aber noch einen Ordner anhängen, der sich nach dem Namen der Anwendung richten sollte. Sie können auch einen Ordner mit dem Namen Ihrer Firma davor setzen.
Es geht nämlich darum, ob Sie in diesem Ordner etwas speichern möchten. Wenn Ja, dann sollten Sie nicht einfach nur den "`Anwendungsdaten`"-Ordner wählen, sondern Sie sollten sich an die übliche Vorgehensweise halten und einen eigenen Ordner für Ihre Anwendung erstellen. So ein Ordner lässt sich ggf. schnell und einfach löschen, während Sie im anderen Fall die Dateien einzeln entfernen und dabei auch Vorsicht walten lassen müssen

```
  UserDirPage.Values[0] := ExpandConstant(
    '{userappdata}\My Company\TestSetup');
end;
```

Wie beim Optionsfenster haben Sie auch hier wieder zwei Möglichkeiten, um die Eingabe bzw. Auswahl des Anwenders zu ermitteln. Entweder Sie benutzen wieder "`NextButtonClick`"

```
  if CurrentPageId = UserDirPage.ID then
    UserFolder := UserDirPage.Values[0];
```

oder Sie verwenden "`CurStepChanged`"

```
  if CurrentStep = ssInstall then
    UserFolder := UserDirPage.Values[0];
```

In beiden Fällen befindet sich der Ordner nun in der `string`-Variablen "`UserFolder`". Um sie in einer Verknüpfung o.ä. nutzen zu können, benötigen wir wieder eine Hilfsfunktion

```
function GetAppDataFolder(dummy: string): string;
begin
  Result := UserFolder;
end;
```

```
[Icons]
Name: "{group}\Test"; FileName: "{app}\MyProg.exe"; \
  WorkingDir: "{code:GetAppDataFolder}"
```

Eine Dateiauswahl

Wenn Sie nach einer Datei suchen wollen oder müssen, dann könnten Sie auch den eben gezeigten Verzeichnisdialog verwenden. Da Sie mit ihm aber nur Ordner auswählen, müssten Sie den Dateinamen immer mit anhängen.
Einfacher geht es allerdings mit einem weiteren Dialog, der speziell für die Suche nach Dateien gedacht ist. Dieser Dialog erfordert eine Variable vom Typ `TInputFileWizardPage`. Für mein Beispiel wird der Dialog die Suche nach PDF-Dateien anbieten.

Zum Erzeugen des Dialogs brauchen wir die Funktion "`CreateInputFilePage`". Wie bereits bei den gezeigten Beispielen bestimmt der erste Parameter die ID der Seite, nach der der Dialog eingefügt werden soll. Die beiden nächsten Parameter definieren den Titel und eine kurze Erläuterung, die im oberen weißen Teil des Setupfensters angezeigt werden. Der letzte `string`-Parameter ist eine etwas ausführlichere Beschreibung im Hauptteil des Fensters

```
  UserFilePage := CreateInputFilePage(wpSelectDir,
    'Pfad zur Inno Setup-Dokumentation wählen',
    'wählt den Pfad zur "inno.pdf" aus',
    'Bitte ..., und klicken Sie dann "Weiter');
```

Die Dialogseite ist nun vorhanden, aber sie ist auch noch leer. Mit Hilfe der "`Add`"-Methode fügen wir nun eine Beschreibung, eine Auswahl von Filtern und eine Standarddateierweiterung in den Dialog ein

```
  UserFilePage.Add('Inno Setup-Dokumentation wählen:',
    'PDF-Dateien|*.pdf|Alle Dateien|*.*',
    '*.pdf');
```

Sie sehen, dass ich zwei Filter benutzt habe: "PDF-Dateien" und "Alle Dateien". So ein Filter besteht immer aus zwei Elementen: der Beschreibung und der dazu gehörenden Dateiendung. Jedes Element wird durch den senkrechten Strich voneinander getrennt.
Wenn Sie Erfahrung mit dem `TOpenDialog` von Delphi haben, dann ist Ihnen dieser Aufbau der Filter bereits bekannt.

Die Eingabe bzw. Auswahl des Anwenders ermitteln Sie nun wie gewohnt, und wie bereits bei der Verzeichnisauswahl auf Seite 32 gezeigt. Dennoch zeige ich Ihnen ein kurzes Beispiel für den bekannten Klick auf den Weiter-Button

```
function NextButtonClick(CurrentId: integer): boolean;
begin
  if CurrentId = UserFilePage.ID then
  begin
    SelectedFile := UserFilePage.Values[0];
    MsgBox(SelectedFile, mbInformation, MB_OK);
  end;

  Result := true;
end;
```

Im Gegensatz zur Verzeichnisauswahl gibt es hier aber keine Prüfung durch das Setup. Hier wird ein Leerstring anstandslos akzeptiert. Wenn die gesuchte Datei für Ihr Setup wichtig ist, dann müssen Sie selbst eine Prüfung vornehmen und leere Angaben unterbinden

```
    s      := UserFilePage.Values[0];          // s : string
    Result := (s <> '') and (FileExists(s));

    if not Result then
      MsgBox('Bitte wählen Sie eine Datei aus!', mbError, MB_OK);
```

Benötigen Sie die Datei innerhalb Ihres Setups, etwa als Parameter für eine Verknüpfung o.ä., dann müssen Sie wieder eine Hilfsfunktion schreiben, die Ihnen den Wert der Variablen als Ergebnis liefert. Das folgende kleine Beispiel geht davon aus, dass Sie die auf der vorigen Seite gezeigte `string`-Variable "`SelectedFile`" verwenden

```
[Icons]
Name: "{group}\Test"; FileName: "{app}\MyProg.exe"; \
  Parameters: "{code:GetSelectedPdfFile}"; WorkingDir: "{app}"


[Code]

function GetSelectedPdfFile(dummy: string): string;
begin
  Result := SelectedFile;
end;

{ ... }
```

Auf das Ende einer Deinstallation warten

In diesem Kapitel geht es um eine kleine Besonderheit beim Entfernen von bereits installierten Anwendungen. Im Normalfall macht man dies über die Systemsteuerung. Manchmal ist es aber auch nötig, eine vorhandene Version direkt aus dem Setup entfernen zu lassen. Das Problem ist aber, dass sich der Uninstaller, wenn er gerade läuft, nicht selbst entfernen kann.
Der Uninstaller greift hier zu einem Trick und kopiert sich selbst in das temporäre Verzeichnis. Die Kopie wird gestartet, und der originale Uninstaller beendet sich. Die Kopie hat vollen Zugriff auf das Programmverzeichnis und kann die Anwendung entfernen.

Dieser Trick erzeugt ein neues Problem für uns. Es ist zwar möglich, ein externes Programm zu starten und auf dessen Ende zu warten, aber in diesem Fall ist es ja nur das Ende des originalen Uninstallers. Die eigentliche Aufgabe übernimmt die Kopie im "`temp`"-Ordner, und dessen Ende wäre interessanter für uns.
Wir müssen also das Verhalten des Uninstallers simulieren. Und das klappt nicht ohne einen Einblick in den Quellcode von Inno Setup, denn so intelligent ist der Uninstaller auch nicht, dass er weiß, wann er das Original und wann er eine Kopie ist. Der tatsächliche Trick ist ein spezieller Parameter, der nur benutzt wird, wenn der originale Uninstaller seine Kopie startet, und der der Kopie klarmacht, dass sie nur eine Kopie ist.

Wir fangen aber mit ein paar Hilfsfunktionen an, die sich auch leicht in anderen Setups verwenden lassen. Da ist zum einen die Funktion zum Ermitteln des Uninstallers einer bereits installierten Version. Im Idealfall können Sie die "`AppId`" des laufenden Setups benutzen. Handelt es sich bei der zu entfernenden Version aber um eine separate Anwendung, dann müssen Sie den korrekten Registryschlüssel angeben

```
function GetUninstaller: string;
begin
  if not RegQueryStringValue(HKLM,
    'Software\Microsoft\Windows\CurrentVersion\Uninstall\???_is1',
    'UninstallString',
    Result) then

  Result := '';
end;
```

Dann sollten Sie bedenken, dass einige Uninstaller mit Parametern aufgerufen werden können. Der ausgelesene Uninstaller besteht also in einigen Fällen nicht nur aus dem Namen der Programmdatei, sondern er kann noch zusätzliche Angaben enthalten.
Beim späteren Start des Uninstallers kann man den ausgelesenen String nicht einfach so an die Funktion übergeben, sondern man muss sie in Programmdatei und Parameter auftrennen. Und das macht die Funktion "`SplitParameters`". Sie erwartet den Namen des Uninstallers als ersten Parameter und liefert als Ergebnis den Dateinamen des Uninstallers ohne Parameter zurück. Die Parameter werden als variabler Parameter im Funktionskopf zurückgegeben

```
function SplitParameters(FileName: string; var Parameters: string):
  string;
var
  i : integer;
  InQuote : boolean;
begin
  Result := '';
  Parameters := '';
  if FileName = '' then exit;

  InQuote := false;
  i := 1;
  while i <= length(FileName) do
  begin
    if (FileName[i] = '"') then InQuote := not InQuote;
    if (FileName[i] = ' ') and (not Inquote) then break;
    i := i + 1;
  end;

  Result := RemoveQuotes(Copy(FileName, 1, i - 1));
  Parameters := Copy(FileName, i + 1, length(FileName));
end;
```

Die nächsten beiden Funktionen kopieren später den Uninstaller in den "`temp`"-Ordner und starten ihn dann von dort

```
function CopyFileToTemp(FileName: string): boolean;
begin
  Result :=
    FileCopy(FileName, Format('%s\%s',
      [ExpandConstant('{tmp}'), ExtractFileName(FileName)]),
    false);
end;

function GetTempName(FileName: string): string;
begin
  Result := Format('%s\%s',
    [ExpandConstant('{tmp}'), ExtractFileName(FileName)]);
end;
```

Soviel dazu. Jetzt zum eigentlichen Hauptteil. Wie ich bereits sagte war hier ein Blick in den Quellcode nötig, denn der Uninstaller startet seine Kopie mit einem Parameter, der sonst nicht verwendet wird. Dieser Parameter heißt "`/SECONDPHASE`" und sorgt dafür, dass die Kopie "weiß", dass sie sich bereits im temporären Ordner befindet. Zusätzlich müssen aber noch Name und Parameter des originalen Uninstallers angegeben werden, damit die Kopie auch die korrekte Anwendung entfernt.

Ich habe dafür eine Funktion namens "`UninstallOldVersion`" geschrieben. Diese Funktion besitzt zwei Parameter. Mit einem können Sie entscheiden, ob mögliche Fehler angezeigt werden. Der zweite erzwingt eine stille Deinstallation.

```
function UninstallOldVersion(const ShowErrorMsg, SilentMode: boolean):
  boolean;
var
  Uninstaller,
  UninstallerParams,
  UninstallerDatFile : string;
  ExecResult : integer;
begin
  Result := true;
```

Also ermitteln wir zunächst einmal den originalen Uninstaller

```
  Uninstaller := SplitParameters(GetUninstaller, UninstallerParams);
```

Wenn die vorhandene Version ohne weitere Rückmeldungen entfernt werden soll, dann können Sie mit Hilfe des "`SilentMode`"-Parameters den so genannten stillen Modus aktivieren. Dabei handelt es sich um einen Parameter für den Uninstaller, der an eventuell schon vorhandene Parameter angehängt wird

```
  if SilentMode then
    UninstallerParams := Format('%s /SILENT', [UninstallerParams]);
```

Der Uninstaller wird dann in den "`temp`"-Ordner kopiert

```
  if (Uninstaller <> '') and (FileExists(Uninstaller)) then
  begin
    if CopyFileToTemp(Uninstaller) then
    begin
```

und sofern auch das erfolgreich war, starten wir diese Kopie mit den eben erwähnten Parametern und warten auf das Ende

```
      Exec(
        GetTempName(Uninstaller),
        Format('/SECONDPHASE="%s" %s', [Uninstaller, UninstallerParams]),
        ExpandConstant('{tmp}'),
        SW_SHOWNORMAL,
        ewWaitUntilTerminated,
        ExecResult);
```

Und das war auch schon das ganze Geheimnis. Die Funktion liefert `TRUE` zurück, wenn das Rückgabeergebnis Null ist. Null steht für `ERROR_SUCCESS` und bedeutet, dass die Ausführung des Programms fehlerfrei beendet wurde

```
      Result := ExecResult = 0;
```

Falls ein Fehler auftrat, kann sich der Anwender darüber informieren lassen. Dies geschieht über den Parameter im Funktionskopf

```
      if (not Result) and (ShowErrorMsg) then
        MsgBox
          (Format('%s (%d)', [SysErrorMessage(ExecResult), ExecResult]),
           mbError,
           MB_OK);
    end else
      Result := false;
  end;
end;
```

Deinstallation von Microsoft Installer-Produkten

Als kleine Ergänzung zur korrekten Deinstallation bereits vorhandener Produkte möchte ich noch den Weg für den Microsoft Installer zeigen. Produkte, die den Microsoft Installer verwenden, nutzen intern eine so genannte *ProductId*, üblicherweise eine eindeutige, einzigartige GUID. Mit Hilfe dieser internen *ProductId* kann die Anwendung gefunden und angepasst werden. Mit ihr kann die Anwendung natürlich auch entfernt werden.

Achtung! Bei der Deinstallation von Produkten, die den Microsoft Installer verwenden, werden im Normalfall Administratorrechte benötigt.

Das Microsoft Installer-API stellt einige Funktionen zur Verfügung, von denen zwei für die gestellte Aufgabe besonders interessant sind

```
function MsiQueryProductState(ProductCode: string): integer;
  external 'MsiQueryProductStateA@msi.dll';

function MsiConfigureProduct(ProductCode: string;
  iInstallLevel: integer; eInstallState: integer): integer;
  external 'MsiConfigureProductA@msi.dll';

const
  INSTALLSTATE_DEFAULT        = 5;
  INSTALLLEVEL_MAXIMUM        = $ffff;
  INSTALLSTATE_ABSENT         = 2;
```

Der Einfachheit halber habe ich zwei Funktionen geschrieben, mit denen diese API-Aufrufe etwas einfacher verwendet werden können

```
function IsMsiProductInstalled(const ProductId: string): boolean;
begin
  Result := MsiQueryProductState(ProductId) = INSTALLSTATE_DEFAULT;
end;

function UninstallMsiProduct(const ProductId: string): boolean;
begin
  Result := false;
  if not IsMsiProductInstalled(ProductId) then exit;

  Result := MsiConfigureProduct(ProductId, INSTALLLEVEL_MAXIMUM,
    INSTALLSTATE_ABSENT) = 0;
end;
```

Diese beiden Funktionen können Sie nun in Ihrem eigenen Skript verwenden. Im Normalfall ist es nun so, dass Sie die *ProductId* des Produktes benötigen, das Sie entfernen möchten. Da es sich in der Regel um eine Anwendung von Ihnen handelt, dürfte das kein Problem darstellen, da Ihnen diese ID bekannt sein muss. Andernfalls wäre es sinnvoll, den Anwender vorher darüber zu informieren, dass auf seinem System ein, mit dem MSI installiertes Produkt gefunden wurde, das Sie entfernen wollen.

Da ich die Produkte von Ihnen bzw. Ihrer Firma natürlich nicht kenne, nehmen wir als Beispiel an, dass Openoffice.org 2.1 entfernt werden soll. Ich habe mich für die 2er-Version entschieden, weil die den Microsoft Installer nutzt. Die *ProductId* finden Sie in der Registry, in dem Schlüssel, in dem auch die Deinstallationsinformationen Ihrer Inno Setup-Setups gespeichert werden. Eigentlich ist der benutzte Registryschlüssel selbst die *ProductId*, aber das muss nicht immer so sein. Es gibt externe Setup-Programme, die als Basis den Microsoft Installer nutzen.

Im Fall von Openoffice können Sie sich aber sowohl am Registryschlüssel selbst, als auch am Eintrag "`UninstallString`" orientieren

```
MsiExec.exe /I{8FB1A5EA-7DA8-4D57-80FB-BD923CCCC852}
```

Da haben wir unsere *ProductId*. Was machen wir jetzt damit? Wir prüfen zuerst, ob auf dem Zielsystem überhaupt Openoffice installiert ist und informieren den Benutzer dann darüber, dass die Anwendung entfernt werden muss

```
const
  OpenofficeOrg2ProductId = '{8FB1A5EA-7DA8-4D57-80FB-BD923CCCC852}';


if IsMsiProductInstalled(OpenofficeOrg2ProductId) then
begin
  if MsgBox('Wollen Sie Openoffice.org 2.1 entfernen?',
    mbConfirmation, MB_YESNO) = IDYES then
  begin
```

Wenn der Anwender diesen Vorgang bestätigt, dann gibt es zwei Möglichkeiten. Man kann einfach nur die Funktion "`UninstallMsiProduct`" aufrufen. Dadurch wird die Anwendung entfernt. Es wäre aber besser, auch das Ergebnis zu kontrollieren. Aus dem Grund bevorzuge ich den folgenden Weg, der a) die besagte Funktion aufruft, und b) danach auch gleich noch einmal prüft, ob das Produkt immer noch installiert ist. Sollte der Anwender die Deinstallation abbrechen, oder sollte etwas anderes nicht funktionieren, würde ein Hinweis erscheinen

```
    if (not UninstallMsiProduct(OpenofficeOrg2ProductId)) or
       (IsMsiProductInstalled(OpenofficeOrg2ProductId)) then
    begin
      MsgBox('Fehler beim Entfernen.', mbError, MB_OK);
    end;
```

Sie sollten an dieser Stelle dann zur Sicherheit auch Ihr Setup abbrechen, wenn es zwingend erforderlich ist, dass die gefundene Anwendung entfernt wird. Informieren Sie den Anwender hier einfach darüber, dass er die Anwendung bitte selbst manuell entfernen soll.

```
  end;
end;
```

Soviel dazu.

Inno Setup Prä-Prozessor

Zu den unterschätzten Fähigkeiten von Inno Setup gehört der Prä-Prozessor, der im QuickStart Pack bereits enthalten ist. Meines Wissens nach kann man ihn auch separat herunterladen und installieren. Und das sollten Sie auch tun, denn seine Vorteile sind nicht zu unterschätzen.
Manchmal wird er aber auch überschätzt. Das resultiert aus der Unkenntnis der Arbeitsweise des Prä-Prozessors. Wenn der Compiler das Setup erstellt, dann hat der ISPP seine Arbeit bereits getan. Der Prä-Prozessor kann beispielsweise nicht dazu benutzt werden, Dateien auf dem Zielrechner des Anwenders auszulesen. Bitte beachten Sie das.

Zeilenumbrüche

Mit einem dieser Vorteile haben Sie bereits im Rahmen dieser Dokumentation Bekanntschaft gemacht: Zeilenumbrüche. Im Normalfall akzeptiert der Compiler nämlich keine. Ich habe auf dem A4-Blatt aber nur eine begrenzte Breite zur Verfügung. Würden Sie meine Anweisungen in Ihr Skript übernehmen, dann würde Sie der Compiler dafür mit Fehlermeldungen bestrafen.
Darum ist es für mich ein großer Vorteil, dass mir der Prä-Prozessor den Zeilenumbruch gestattet, wenn ich einen Backslash \ benutze. Der Prä-Prozessor formatiert das Skript vor dem eigentlichen Kompilieren wieder um und entfernt solche Zeilenumbrüche. Dadurch hat der Compiler keinen Grund, sich zu beschweren, und ich erreiche eine bessere Lesbarkeit, weil ich das Skript nicht noch horizontal scrollen muss.

Symbole

Die Schreibarbeit in Ihrem Setup kann durch so genannte Symbole reduziert werden. Nehmen Sie die folgenden Beispielzeilen eines deutschsprachigen Setups

```
[Setup]
AppName=DelphiHTML²
AppVerName=DelphiHTML²
DefaultDirName={pf}\DelphiHTML
DefaultGroupName=DelphiHTML
OutputBaseFilename=DelphiHTML-setup
OutputDir=.
```

Sie sehen, dass der Name der Anwendung recht häufig vorkommt. Nun stellen Sie sich weiter vor, Sie benötigen den Namen noch für Verknüpfungen, Registryeinträge usw. Es besteht die Gefahr, dass sich dabei doch einmal ein Tippfehler einschleicht. Ein Buchstabendreher. Etwas in der Art. Und schon heißt der Registryschlüssel anders als gewollt, oder die Verknüpfungen landen in einem falschen Ordner.
Oder stellen Sie sich vor, Sie müssten den Namen der Anwendung ändern. Dann müssten Sie das ganze Skript durchsuchen und jedes Vorkommen des Namens entsprechend ersetzen.

Es wäre also weitaus einfacher, wenn man den Namen oder bestimmte Ordnerbezeichnungen nur ein einziges Mal festlegt und im ganzen Skript darauf zugreifen kann. Und genau das funktioniert mit den Symbolen. Wie Sie sehen unterscheiden sich der Name des Ordners und der Name der Anwendung nur durch die kleine hochgestellte Zwei. Sie könnten also ein Hauptsymbol definieren, das als Grundlage für Ordner und evtl. Registryeinträge o.ä. dient. Und von ihm leiten sie ein zweites Symbol für den Anwendungsnamen ab.

```
#define BASENAME "DelphiHTML"
#define APPNAME BASENAME + "²"
```

Beide Symbole können Sie nun in Ihrem Skript verwenden, indem Sie sie in geschweifte Klammern setzen und vor dem Namen das Doppelkreuz-Zeichen # ergänzen.

```
[Setup]
AppName={#APPNAME}
AppVerName={#APPNAME}
DefaultDirName={pf}\{#BASENAME}
DefaultGroupName={#BASENAME}
OutputBaseFilename={#BASENAME}-setup
OutputDir=.
```

Beim Kompilieren schaltet sich nun der Prä-Prozessor dazwischen, ersetzt die Symbole durch die zugeordneten Werte, und für den Compiler entsteht ein völlig normales Skript, das er in das Setup übersetzt. Für Sie verringert sich der Arbeitsaufwand beim Erstellen und Anpassen des Skriptes.

Achtung! Ich habe nicht ohne Grund am Anfang erwähnt, dass es sich bei diesem Beispiel um ein deutschsprachiges Setup handelt. Bei einem mehrsprachigen Setup können Sie Ordnernamen, Registryeinträge u.ä. nach dem gezeigten Muster per Symbol definieren. Für zu lokalisierende Bezeichnungen, etwa dem Anwendungsnamen, Namen von Verknüpfungen o.ä., sollten Sie wie gewohnt die Sektion "`[CustomMessages]`" verwenden (vgl. Seite 16ff).

Natürlich können Sie solche Symbole auch verwenden, um verschiedene Versionen des Setups zu erstellen. Sagen wir als Beispiel, Sie möchten eine kostenlose Version Ihrer Anwendung anbieten, die auch nur Ihr Programm und die entsprechenden Dateien enthält. Außerdem wollen Sie aber auch eine kostenpflichtige Version anbieten, die neben der Anwendung auch noch den Quellcode enthält. Mit Hilfe der Symbole können Sie das alles in einem einzigen Skript realisieren.

Gehen wir von der folgenden recht einfachen "`[Files]`"-Sektion aus

```
[Files]
Source: "*"; DestDir: "{app}"
Source: "source\*"; DestDir: "{app}\source"; Flags: recursesubdirs
```

Wie gesagt, soll der Quellcode nur den zahlenden Kunden zur Verfügung stehen. Definieren Sie daher ein Symbol mit einem zur Aufgabe passenden Namen, damit Sie auch später noch erkennen können, welchem Zweck es dient. Zum Beispiel

```
#define FullVersion
```

Wenn Sie sich mit der bedingten Kompilierung in Programmiersprachen auskennen, werden Sie die folgenden Erläuterungen nicht weiter überraschen, denn das Prinzip ist identisch. Mit Hilfe von `#ifdef`, `#else` und `#endif` können Sie im Skript dafür sorgen, dass bestimmte Zeilen nur berücksichtigt werden, wenn ein bestimmtes Symbol vorhanden ist. Um bei unserem Beispiel zu bleiben

```
[Files]
Source: "*"; DestDir: "{app}"
#ifdef FullVersion
Source: "source\*"; DestDir: "{app}\source"; Flags: recursesubdirs
#endif
```

würde bedeuten, dass der Quellcode Ihrer Anwendung nur ins Setup integriert werden würde, wenn das Symbol "`FullVersion`" zuvor definiert wurde. Wenn Sie das Symbol auskommentieren oder entfernen, würde die Zeile zwischen `#ifdef` und `#endif` ignoriert werden.

Wenn Sie eine Alternative benötigen, dann benutzen Sie zusätzlich `#else`. Das eben gezeigte Beispiel geht davon aus, dass das Symbol vorhanden ist. Wenn Ja, wird die Anweisung kompiliert. Wenn Nein, dann wird sie ignoriert. Eine Alternative würde beide Fälle berücksichtigen. Wenn das

Symbol vorhanden ist, reagiert das Skript auf eine bestimmte Art. Fehlt das Symbol, reagiert es auf eine andere Art.
Das ließe sich zum Beispiel recht elegant beim Namen der Setupdatei anwenden. Um Ihre beiden Setupversionen auch anhand des Namens voneinander unterscheiden zu können, nennen Sie die kostenlose Version einfach nur "`setup.exe`". Die kostenpflichtige Version dagegen könnte etwa "`setup-pro.exe`" heißen

```
[Setup]
#ifdef FullVersion
OutputBaseFilename=setup-pro
#else
OutputBaseFilename=setup
#endif
```

Aufgaben (= Tasks)

Nach dem kleinen Ausflug der vorangegangenen Seiten wollen wir uns nun den so genannten *Tasks* widmen. Dabei handelt es sich um Aufgaben, die das Setup anbietet, und die auch mit der Installation in irgendeiner Form zu tun haben, deren Ausführung aber im Ermessen des Anwenders liegt. Es ähnelt ein bisschen der Komponentenauswahl bei der benutzerdefinierten Installation.

Zur Verdeutlichung möchte ich Sie an die Verknüpfungen (s. Seite 13*ff*) erinnern. Als Anwender erwartet man, dass sich ein Programm im Startmenü einträgt, damit man es einfach aufrufen kann. Wenn das Programm zusätzlich auch eine Verknüpfung auf dem Desktop anlegt, dann ist das zwar auch hilfreich, aber es muss nicht unbedingt im Interesse aller Anwender sein.
Hier bieten sich dann die *Tasks* an, mit deren Hilfe jeder Anwender für sich selbst entscheiden kann, ob er die entsprechende Aktion ausführen möchte oder nicht. Bleiben wir doch gleich bei der Verknüpfung auf dem Desktop.

Um einen Task anzulegen, benötigen wir die Sektion "`[Tasks]`" mit folgendem Inhalt

```
[Tasks]
Name: desktopicon; \
  Description: "Verknüpfung auf dem Desktop erstellen"; \
  Flags: unchecked
```

Name	Erklärung
`Name`	frei wählbarer interner Name
`Description`	Beschreibung, die in Inno Setup angezeigt wird
`GroupDescription`	Beschreibung für mehrere Tasks
`Flags`	spezielle Einstellungen (s. Hilfe)

Zur Verknüpfung selbst gibt es nichts weiter zu sagen. Es wird lediglich ein neuer Parameter ergänzt, der auf den internen Tasknamen verweist

```
[Icons]
Name: "{userdesktop}\TestSetup"; Filename: "{app}\MyProg.exe"; \
  WorkingDir: "{app}"; Tasks: desktopicon
```

Damit wird die Verknüpfung auf dem Desktop jetzt nur noch angelegt, wenn der Anwender den entsprechenden *Task* aktiviert - sprich: das Häckchen in die Auswahlbox setzt.

Tasks programmgesteuert deaktivieren

Diese *Tasks* sind standardmäßig aktiv, es sei denn, Sie nutzen das oben gezeigte `unchecked`-Flag. Was ist aber, wenn so eine Aufgabe von einer bestimmten Bedingung des Systems abhängig gemacht werden soll?
Eine Möglichkeit wäre vielleicht der "`Check`"-Parameter (s. Seite 29), obwohl er eigentlich nur die Ausführung einer Aufgabe verhindert, wenn die Bedingung nicht erfüllt ist. Er sorgt aber nicht dafür, dass die Aufgabe deaktiviert wird, so dass sie der Anwender erst gar nicht auswählen kann.

Hier kommt nun wieder die Skriptsprache ins Spiel. Mit ihr können Sie bestimmte Bedingungen überprüfen und ggf. den dazu gehörenden `Task` deaktivieren. Wir wollen davon ausgehen, dass wir in der Registry nach einem Eintrag suchen, der auf einen bestimmten Pfad auf dem Zielrechner verweist. Wenn dieser Pfad nicht vorhanden ist, soll die entsprechende Aufgabe des Setup deaktiviert werden, so dass sie zwar sichtbar ist, aber nicht angeklickt werden kann.

Das Beispiel stammt aus der Testversion eines Setups, das ich in dieser Form nie veröffentlicht habe. Ich wollte nur prüfen, ob und wie es mit Inno Setup geht. Es handelt sich dabei um die, Ihnen vielleicht sogar bekannten Win32-API-Tutorials für Delphi, einer Sammlung von Hilfedokumenten, die sich mit der nonVCL-Programmierung beschäftigen. Diese Dokumente können in die Hilfe von Borland Delphi 2005 und in das Platform SDK von Microsoft integriert werden.

Ich demonstriere es am zweiten Beispiel, dem PSDK

```
[CustomMessages]
PSDKPlugin=Hilfedokumente in Microsofts PSDK integrieren

[Tasks]
Name: psdkplugin; Description: "{cm:PSDKPlugin}"; Flags: unchecked
```

Jetzt benötige ich eine Ereignisfunktion von Inno Setup, in der ich prüfe, welche Dialogseite gerade aktuell ist. Mich interessiert die Task-Auswahl, *wpSelectTasks*, dabei besonders. Die Funktion heißt "`CurPageChanged`". Sie hat nur einen Parameter, und der identifiziert die aktuelle Seite

```
[Code]
procedure CurPageChanged(const iCurPage: integer);
var
  res  : boolean;
  psdk : string;
  i    : integer;
begin
  if(iCurPage = wpSelectTasks) then
  begin
```

An dieser Stelle muss ermittelt werden, ob das Platform SDK installiert ist. Dazu greife ich auf die Registry zu und ermittle einen bestimmten Wert. Da dieser Wert ein Verzeichnis ist, prüfe ich mit der Funktion "`DirExists`", ob dieses Verzeichnis auch existiert

```
    res := (RegQueryStringValue(HKEY_LOCAL_MACHINE,
              'Software\Microsoft\MicrosoftSDK\Directories',
              'Install Dir',
              psdk)) and
           (psdk <> '') and
           (DirExists(psdk));
```

Wenn mindestens eine der hier gezeigten 3 Bedingungen `FALSE` ist, dann ist auch die benutzte `bool`-Variable `FALSE`, und Sie können davon ausgehen, dass das Platform SDK nicht vorhanden ist, bzw. nicht zweifelsfrei gefunden werden konnte.
Deshalb greifen wir nun auf die Liste mit den Aufgaben zu und wollen die eine Aufgabe deaktivieren, die für die Integration der Hilfedokumente ins PSDK zuständig ist. Die Liste mit den Aufgaben wird durch das "`TasksList`"-Objekt zur Verfügung gestellt. Dabei handelt es sich, laut Deklaration in der Hilfe, um eine Art Liste, deren Einträge über die Eigenschaft "`Items`" abgefragt werden können.
Wenn Sie Erfahrung mit Delphi besitzen, wird Ihnen der Begriff `TStrings` bekannt sein. Darum handelt es sich hier nämlich. Und wie in Delphi stellt `TStrings` die Funktion "`IndexOf`" zur Verfügung, mit der Sie nach einem Eintrag suchen können.
Diese Funktion erwartet als Parameter den gesuchten Text und liefert dafür seinen Index innerhalb der Liste zurück. Im Fehlerfall, das heißt, wenn es den Eintrag gar nicht gibt, ist der Rückgabewert -1.

Nun könnte man es sich einfach machen und den Text 1:1 angeben. Wenn Sie aber ein mehrsprachiges Setup haben, dann müssen Sie auch alle Sprachen berücksichtigen. Das heißt, Sie müssen eigentlich nicht, weil Sie ja mit der schon erwähnten Funktion "`ExpandConstant`" den gewünschten String aus der "`[CustomMessages]`"-Sektion ermitteln können, und das Ergebnis ist ja immer der String in der vom Anwender ausgewählten Setupsprache.

Lassen Sie uns also davon ausgehen, dass das Platform SDK nicht gefunden wurde. Mit Hilfe der genannten Funktionen suchen wir nun in der Liste der Aufgaben nach dem *Task*, der für das PSDK zuständig ist

```
    if(not res) then
    begin
      i := WizardForm.TasksList.Items.IndexOf(
        ExpandConstant('{cm:PSDKPlugin}'));
```

Die Variable "`i`" enthält nun den Index des *Tasks*. Und mit Hilfe der Eigenschaft "`ItemEnabled`", die von uns diesen Index erwartet, können wir den *Task* nun deaktivieren. Vorausgesetzt, der Index ist nicht -1, denn das deutet, wie schon erwähnt, darauf hin, dass der String nicht gefunden wurde

```
      if(i <> -1) then
        WizardForm.TasksList.ItemEnabled[i] := false;
    end;
  end;
end;
```

Und so würde das Ergebnis aussehen, wenn Sie das Setup auf einem Rechner starten, auf dem das PSDK nicht installiert ist

Eine Express-Installation

Auf den vorigen Seiten haben Sie sich mit einem typischen Setup beschäftigt.
Aber mal ehrlich: läuft es denn wirklich so ab?
Man kann ohne Übertreibung sagen, dass in 95% aller Fälle die meisten Seiten eines Setups einfach weggeklickt werden. Und das ist aus Sicht des Anwenders auch nachvollziehbar. Man verlässt sich darauf, dass die Voreinstellungen sinnvoll sind, um mit dem zu installierenden Programm ohne Probleme arbeiten zu können.

Abfrage des Zielverzeichnisses? Irgendwo im "`Programme`"-Ordner.
Abfrage der Gruppe im Startmenü? Die Vorgabe scheint sinnvoll.
Auswahl der zu installierenden Komponenten? Lassen wir die Haken so drin. Passt schon.
Zusätzliche Aufgaben? Desktop-Verknüpfung? Okay.

In dem Fall kann man die Seiten auch gleich verbergen, die der Anwender sowieso wegklicken würde. Die Auswahl kann dem Anwender mit Hilfe einer Auswahlbox auf der Startseite erleichtert werden.

```
var
  cb1 : TCheckBox;

procedure InitializeWizard;
begin
  cb1 := CreateCheckBox(WizardForm.WelcomeLabel2.Left,
    WizardForm.WelcomeLabel2.Top + WizardForm.WelcomeLabel2.Height - 30,
    120,
    CustomMessage('MsgExpressInstall'),
    WizardForm.WelcomePage,
    nil);
end;
```

Benutzt wird hier eine Funktion, die eine Checkbox auf einer bestimmbaren Seite erzeugt und dafür als Parameter die Position, die Breite, sowie natürlich Beschriftung und das Klick-Ereignis erwartet. Letzteres kann hier bei vorerst ignoriert werden

```
function CreateCheckbox(const X, Y, Width: integer; Caption: string;
  Parent: TWinControl; ClickEvent: TNotifyEvent): TCheckBox;
begin
  Result := TCheckBox.Create(WizardForm);
  if Result <> nil then
  begin
    Result.Left    := X;
    Result.Top     := Y;
    Result.Width   := Width;
    Result.Caption := Caption;
    Result.Checked := false;
    Result.Parent  := Parent;
    Result.OnClick := ClickEvent;
  end;
end;
```

Das Ergebnis kann sich sehen

Jetzt muss nur noch dafür gesorgt werden, dass bei Auswahl der Express-Installation die nicht notwendigen Seiten verschwinden. Dabei hilft die "`ShouldSkipPage`"-Funktion

```
function ShouldSkipPage(PageId: integer): boolean;
begin
  case PageId of
    // Auswahl des Zielverzeichnisses
    wpSelectDir,
    // Auswahl der Programmgruppe im Startmenü
    wpSelectProgramGroup,
    // Auswahl der zu installierenden Komponenten
    wpSelectComponents,
    // Auswahl der zusätzlichen Aufgaben
    wpSelectTasks,
    // Anzeige der Zusammenfassung vor der Installation
    wpReady:
      Result := cb1.Checked;
    else
      Result := false;
  end;
end;
```

Wenn also angenommen wird, dass noch eine Lizenz angezeigt werden soll, dann reduziert sich die Klickarbeit des Anwenders enorm. Er klickt auf der Startseite die Express-Installation an, liest die Lizenz, und der nächste Klick startet bereits die Installation.

Ein kleiner Makel bleibt: der "Weiter"-Button heißt immer noch "Weiter"-Button. Es ist sinnvoll, auf der Lizenzseite die Beschriftung "Installieren" zu zeigen, wenn die Express-Installation gewählt wurde. Der Anwender soll schließlich wissen, dass dies sein letzter Schritt vor dem Kopieren der Dateien und dem Einrichten des Programms ist. Korrigieren lässt sich das mit der Funktion "`CurPageChanged`"

```
procedure CurPageChanged(CurrentPageId: Integer);
begin
  if CurrentPageId = wpLicense then
  begin
    if cb1.Checked then
      WizardForm.NextButton.Caption := SetupMessage(msgButtonInstall)
    else
      WizardForm.NextButton.Caption := SetupMessage(msgButtonNext);
  end;
end;
```

Damit ist auch das Problem gelöst.

Ich möchte aber noch einmal anmerken, dass das Setup zuvor in den Standardeinstellungen getestet werden sollte. Nur so kann man herausfinden, ob wirklich alle nötigen Komponenten installiert werden, die zum reibungslosen Gebrauch des Programms erforderlich sind. Andernfalls ist das Skript noch anzupassen.

Upgrades

Was sind denn Upgrades? Grundsätzlich auch erst einmal normale Setups. Ein Upgrade ist aber so ausgelegt, dass es eine bereits vorhandene ältere Version erkennen und ersetzen kann. Mir sind schon Setups aufgefallen, die diesen Punkt vernachlässigen. Wurde dann eine neue Version veröffentlicht, konnte man in einer beiliegenden Textdatei oder auf der Homepage der Entwickler lesen, dass man bitte vorher die alte Version entfernen soll.

Das lässt sich aber auch automatisieren, sofern man eine ganz einfache Regel beachtet. Inno Setup benutzt intern die Variable "`AppId`" als Basis für den Deinstallationsschlüssel in der Registry. Wenn Sie sich das gezeigte Beispiel in dieser Dokumentation anschauen, werden Sie merken, dass diese Variable nirgendwo vorkommt.
In so einem Fall benutzt der Compiler den Wert von "`AppName`" und ordnet ihn der besagten Variablen zu. Später wird noch "`_is1`" angehangen und als Schlüsselname für die Registry benutzt. Würde Ihre Angabe zum Beispiel

```
[Setup]
AppName=TestSetup
```

lauten, dann würde der Schlüssel demzufolge "`TestSetup_is1`" heißen. Für ein funktionierendes Upgrade müssten Sie also den Wert von "`AppName`" in jeder noch folgenden Setupversion beibehalten. Oder Sie wählen den einfachsten Weg und legen diesen oder einen anderen neutralen Wert gleich auf "`AppId`" fest

```
[Setup]
AppId=TestSetup
```

Neutral bedeutet, dass dieser Name möglichst keine Versionsangaben oder ähnliches enthalten sollte. Ideal wäre eine so genannte GUID im Format

```
AppId={{aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee}
```

weil hier die Wahrscheinlichkeit am geringsten ist, dass ein anderes Setup den gleichen Wert benutzt. Zum Erzeugen von GUIDs gibt es diverse Tools. Einige Programmiersprachen haben auch eine eingebaute Funktion dafür.

Ich möchte meine Hand für die Einzigartigkeit zwar nicht ins Feuer legen, aber bisher habe ich noch keine Probleme entdecken können. Wenn Sie etwa eine eingebaute Netzwerkkarte haben, dann ist das letzte Feld bei jeder automatisch generierten GUID immer identisch, weil es sich dabei um die MAC-Adresse bzw. eine Kodierung derselben handelt. Schon das macht es an sich unmöglich, dass noch jemand die gleichen GUIDs generiert. Ohne Netzwerkkarte wird das letzte Feld nach einem Zufallsmuster errechnet. Aber auch hier würde die Chancen so einschätzen, dass Ihre GUIDs einzigartig sind.

Wie funktioniet das Upgrade nun? Wir nehmen an, dass eine bereits ältere Version installiert ist. Das neue Setup startet und findet diese alte Version, weil es dieselbe ID benutzt. Durch die internen Voreinstellungen von Inno Setup wird nun die bereits vorhandene LOG-Datei der alten Version ergänzt. Das Setup vermerkt also nun nur noch zusätzliche Aufgaben, die für die neue Version relevant sind. Etwa: welche neuen Dateien wurden installiert? welche neuen Verknüpfungen wurden angelegt? usw.

Zusätzlich wird der Name der alten Version im Softwaremodul in der Systemsteuerung durch den neuen Namen ersetzt.

Die alten Einstellungen ignorieren

Wenn Sie das Upgrade ausprobieren, werden Sie vielleicht feststellen, dass Inno Setup weiß, welche Komponenten Sie beim Original installiert haben, usw. Das liegt an den folgenden Direktiven, die standardmäßig auf *yes* gesetzt sind

Name	Erklärung
UsePreviousAppDir	Den Pfad des Originals verwenden
UsePreviousGroup	Die Programmgruppe des Originals verwenden
UsePreviousSetupType	Die installierten Komponenten ermitteln und anzeigen
UsePreviousTasks	Die erledigten Aufgaben ermitteln und anzeigen
UsePreviousUserInfo	Die Benutzerdaten des Originals verwenden

Im Klartext heißt das, sobald eine installierte Version gefunden wird, ermittelt Inno Setup alle relevanten Dinge. Haben Sie bspw. nur die erste Komponente installiert, zeigt Ihnen das neue Setup auch nur diese Komponente als ausgewählt an. Haben Sie eine Desktopverknüpfung erstellt, ist auch diese Aufgabe im Upgrade aktiv, usw.

Möchten Sie das umgehen, sollten Sie zumindest die beiden fett markierten Werte auf *no* setzen

```
[Setup]
UsePreviousSetupType=no
UsePreviousTasks=no
```

Auf diese Weise ignoriert das Setup die Komponenten und die *Tasks* und bietet Ihnen eine "frische Auswahl" an. ☺

Versionskontrolle

Damit besteht eigentlich nur noch ein Problem: Zwar funktioniert es wunderbar, eine neue Version einer Anwendung über eine alte installieren zu lassen, aber wer garantiert, dass es sich in jedem Fall um eine neue Version handelt? Weil nun jede Version die gleiche "AppId" verwendet, würde selbstverständlich auch eine ältere Version über eine bereits installierte neuere installiert werden können.
Um das zu verhindern, möchte ich Ihnen im folgenden eine Lösung zeigen, bei der die jeweils aktuelle Versionsnummer Ihrer Anwendung auf dem Zielsystem gespeichert wird. Diese Nummer wird von jedem Setup beim Starten überprüft. Stellt das Setup fest, dass die gespeicherte Version aktueller ist, bricht es das Setup mit einer entsprechenden Hinweismeldung ab. Für diese Lösung benötigen Sie aber sowohl den Prä-Prozessor, als auch die Skriptsprache.

Zum Speichern der Versionsnummer habe ich mich für den Uninstall-Schlüssel der Anwendung entschieden, weil er sowieso bei 95% aller Setups angelegt wird. Um nach Möglichkeit Tippfehler zu vermeiden, habe ich zwei Symbole deklariert. Beim ersten Symbol handelt es sich um die "AppId", die Sie in jeder Setupversion beibehalten sollten, wie ich auch zu Anfang dieses Kapitels erläutert habe. Das zweite Symbol ist für die spätere Verwendung in der Skriptsektion gedacht und definiert den Pfad des Uninstall-Schlüssels, der sich u.a. auch aus der "AppId" ergibt

```
#define APPID "TestSetup"
#define UNINSTKEY \
  "Software\Microsoft\Windows\CurrentVersion\Uninstall\" + \
  APPID + "_is1"

[Setup]
AppId={#APPID}
```

Jetzt benötigen wir die Versionsnummer. Auch diese wird mit dem Prä-Prozessor festgelegt. Auf den Grund komme ich sofort zu sprechen, zuvor möchte ich mich aber dem Aufbau der Nummer widmen. Üblich sind Versionsnummer mit vier Stellen, bspw. `1.0.0.0`. Diese vier Stellen werden aber getrennt deklariert, damit man später einfacher damit rechnen kann, und damit man die Nummer einfacher aktualisieren kann

```
#define MAJOR 1
#define MINOR 0
#define RELEASE 0
#define BUILD 0
```

Und damit wäre ich auch schon beim Grund für die Verwendung des Prä-Prozessors. Es gibt in Ihrem Skript sicher ein paar Stellen, an denen Sie die Versionsnummer des Setups angegeben haben. Beispielsweise hier

```
[Setup]
AppVerName=TestSetup 1.0.0.0
AppVersion=1.0.0.0
```

Mit jeder neuen Version müssten Sie nun solche Stellen suchen und die Versionsnummer an Ihre neue Version anpassen. Der Prä-Prozessor verringert die Arbeit, wenn Sie die Versionsnummer einmal, wie oben gezeigt, als Symbole deklarieren und dann auf diese Symbole zugreifen. Weil es sich bei den vier Symbolen aber um `integer`-Werte handelt, sollten Sie ein weiteres Symbol deklarieren, dass die vier Stellen in einen typischen Versionsstring umwandelt.

```
#define APPVER Str(MAJOR) + "." + Str(MINOR) + "." + \
  Str(RELEASE) + "." + Str(BUILD)

[Setup]
AppVerName=TestSetup {#APPVER}
AppVersion={#APPVER}
```

Der Vorteil dürfte klar sein. Wenn Sie die neue Version `1.1.0.0` veröffentlichen wollen, dann ändern Sie nur das entsprechende Symbol. Wenn Sie das Skript neu kompilieren, sorgt der Prä-Prozessor dafür, dass dann überall dort die aktuelle Versionsnummer erscheint, wo Sie das Symbol "`APPVER`" benutzt haben.

Das Speichern der Versionsnummer passiert nach einem bestimmten Prinzip. Ich habe jeweils zwei Werte zu einem so genannten `dword` zusammengefügt. Ein `dword` ist ein 32-Bit-Wert, die beiden Felder teilen sich also jeweils 16 Bit.

```
00 00 00 01 00 00 00 00
```

Damit ergibt sich pro Feld ein Maximalwert von `0xffff`, womit Sie pro Feld Versionsnummern von 0 bis 65535 bilden können. Das sollte also erst einmal für ein paar Setups genügen. Das gezeigte Beispiel kodiert übrigens die Haupt- und die Nebenversion `1.0`. Eine einfache Addition genügt nicht, wie Sie sich vielleicht denken können. Sie müssen dafür sorgen, dass die Hauptversion in die oberen 2 Bytes des `dword`-Wertes verschoben werden. Und das lässt sich mit der folgenden kleinen Funktion recht einfach realisieren

```
[Code]
function CreateDWord(const Hi, Lo: word): dword;
begin
  Result := (Hi shl 16) or Lo;
end;
```

Sie übergeben dieser Funktion jeweils zwei Felder der Versionsnummer. Beispielsweise eben die Haupt- und die Nebenversionsnummer. Intern wird der "`Hi`"-Wert um 16 Bit nach links verschoben. Aus einer Zahl wie `0x0001` wird dadurch `0x00010000`. Sie könnten auch einfach den Wert

0x10000 multiplizieren. Der "Lo"-Wert kann danach addiert werden. In der gezeigten Funktion geschieht das per or, ein simples Pluszeichen ginge aber ebenso.
Diese Funktion rufen wir nun in der eigentlichen Prüffunktion auf. Sie soll die aktuelle Version des Setups mit der im System gespeicherten Nummer vergleichen.

```
function IsSetupNewer: boolean;
var
  SetupMajor,
  SetupMinor,
  SavedMajor,
  SavedMinor    : dword;
  tmp           : string;
begin
```

Wir müssen aber damit rechnen, dass es noch gar keine gespeicherten Informationen gibt, weil das laufende Setup die allererste Version ist. Ebenso könnte es sich aber auch um eine installierte alte Version handeln, die noch nicht über diese Versionskontrolle verfügt

```
  if(not RegQueryDWordValue(HKLM,'{#UNINSTKEY}','MajorVer',
       SavedMajor)) or
    (not RegQueryDWordValue(HKLM,'{#UNINSTKEY}','MinorVer',
       SavedMinor)) then
```

Sie sehen hier übrigens, dass in diesen zwei Anweisungen auch gleich auf das eingangs definierte Symbol "UNINSTKEY" zugegriffen wird. Für den zweiten Fall müssen wir noch vorsorgen. Wenn es sich um eine alte Version ohne Kontroll handelt, ermitteln wir ggf. den Uninstaller

```
    if(RegQueryStringValue(HKLM,'{#UNINSTKEY}',
         'UninstallString',tmp)) and
      (tmp <> '') and
      (fileexists(tmp)) then
    begin
```

Wird ein existierender Uninstaller gefunden, dann müssen Sie davon ausgehen, dass eine Version ohne Kontrolle installiert ist. In dem Fall zeigen wir eine Meldung an, dass die installierte Version nicht zweifelsfrei identifiziert werden konnte. Es könnte eine ältere Version sein, es könnte aber auch eine aktuellere Version sein. Der Anwender hat dann die Wahl, ob er das Setup fortsetzen möchte oder nicht

```
      Result := (MsgBox(
        ExpandConstant('{cm:NotVerifiedVersionFound}'),
        mbConfirmation,MB_YESNO or MB_DEFBUTTON2) = IDYES);
```

Da es aber dennoch keine gespeicherten Versionsinformationen gibt, können wir die Funktion an dieser Stelle auch verlassen

```
      exit;
    end;
  end;
```

Wenn die gespeicherten dwords gefunden worden sind, dann können wir aus der aktuellen Versionsnummer des laufenden Setups ebenfalls zwei dwords erzeugen. Wie gesagt, es werden zwei Felder zusammengefasst: Haupt- und Nebenversion, sowie Release und Build

```
  SetupMajor    := CreateDWord({#MAJOR},{#MINOR});
  SetupMinor    := CreateDWord({#RELEASE},{#BUILD});
```

Das Setup soll nun nur fortgesetzt werden, wenn es aktueller ist als die schon installierte Version. Dazu muss der erste dword-Wert "SetupMajor" größer sein als der gespeicherte Wert "SavedMajor"

```
  Result := (SetupMajor > SavedMajor) or
```

Ansonsten darf "SetupMajor" zwar mit "SavedMajor" identisch sein, aber in dem Fall muss dann "SetupMinor" größer sein als "SavedMinor". "SetupMinor" darf aber auch mit "SavedMinor" identisch sein, weil es sich dann um das Setup der schon installierten Version handelt, und der Anwender vermutlich fehlende Komponenten nachinstallieren möchte

```
    ((SetupMajor = SavedMajor) and (SetupMinor >= SavedMinor));
```

In keinem Fall darf aber einer der Werte des Setups, egal ob "SetupMajor" oder "SetupMinor" kleiner sein als sein gespeichertes Gegenstück. Wenn doch, dann ist das Setup offenbar älter als die schon installierte Version, und wir wollen eine Meldung ausgeben

```
  if(not Result) then
    MsgBox(ExpandConstant('{cm:NewerAppFound}'),mbError,MB_OK);
end;
```

```
[CustomMessages]
de.NewerAppFound=Es ist bereits eine aktuellere Version installiert.
de.NotVerifiedVersionFound={#APPNAME} ist vermutlich schon \
  installiert worden, aber Setup kann die Version nicht \
  zweifelsfrei ermitteln.%nEs besteht die Möglichkeit, dass \
  dieses Setup älter ist als die bereits installierte \
  Version.%nWollen Sie das Setup trotzdem und auf eigenes \
  Risiko fortsetzen?
```

Wie funktioniert es? Ich möchte es am Beispiel der beiden Versionen 1.0.0.0 und 1.1.0.0 erklären. Die Version 1.0 soll dabei die installierte Version sein, 1.1 wäre das laufende Setup. Beim Kodieren der Haupt- und Nebenversion entstehen nun die folgenden Werte

```
1.0 = 00 00 00 01 00 00 00 00 = 65536
1.1 = 00 00 00 01 00 00 00 01 = 65537
```

Die kodierten Werte werden nun völlig normal als numerische Werte gelesen und auch als solche miteinander verglichen. Und wenn Sie 65536 und 65537 vergleichen, dann ist der zweite Wert natürlich größer. Das bedeutet, das Setup wurde als aktueller identifiziert, weil die oben genannte erste Bedingung bereits zutrifft. Die Minorwerte können in dem Fall vernachlässigt werden, weil beide Null sind.

Etwas anders wäre das bei den Versionen 1.0.0.0 und 1.0.0.1. Hier ergibt sich in beiden Fällen ein Majorwert von 65536, weil Haupt- und Nebenversion jeweils 1.0 sind

```
1.0 = 00 00 00 01 00 00 00 00 = 65536
```

Damit würde die erste Bedingung nicht mehr zutreffen, denn hier wären der Majorwert des Setups und der gespeicherte Majorwert identisch. Es muss also die zweite Bedingung zutreffen. Der Minorwert des Setups muss größer sein als der gespeicherte Wert, oder zumindest gleich, damit das Setup als aktueller gilt

```
0.0 = 00 00 00 00 00 00 00 00 = 0
0.1 = 00 00 00 00 00 00 00 01 = 1
```

Vergleichen Sie nun Null und Eins, und Sie werden mir zustimmen, dass die Eins größer ist. Das Setup ist also auch in diesem Fall aktueller.

Das Durchrechnen der umgekehrten Situation halte ich an dieser Stelle für unnötig, weil Sie einfach nur die beiden gezeigten Beispiele aus der anderen Perspektive betrachten müssen. Ich bin davon ausgegangen, dass das Setup die höhere Versionsnummer benutzt. Drehen Sie diesen Ansatz um, dann müssten Sie die andere Wirkung der Prüffunktion (Meldung, dass eine aktuellere Version der Anwendung bereits installiert ist) belegen können.

Wie dem auch sei, Sie müssen die gezeigte Funktion "`IsSetupNewer`" noch aufrufen. Ideal ist dafür die "`InitializeSetup`"-Funktion, weil Sie mit ihrem Rückgabewert darüber entscheiden können, ob das Setup fortgesetzt werden soll oder nicht.
Und die Prüffunktion liefert das passende Ergebnis zurück, so dass Sie es einfach und elegant an die Ereignisfunktion übergeben können. Wenn das Setup aktueller ist, oder immerhin noch die gleiche Version wie die installierte, dann ist das Ergebnis `TRUE`, und das Setup startet. Handelt es sich um eine ältere Version, dann ist das Ergebnis `FALSE`, und das Setup wird abgebrochen

```
function InitializeSetup: boolean;
begin
  Result := IsSetupNewer;
end;
```

Wenn nun das Setup aktueller ist, und wenn der Anwender die Installation durchgeführt hat, dann muss natürlich die aktuelle Versionsnummer in die Registry geschrieben werden. Ansonsten wäre die ganze Idee recht nutzlos. Oder sagen wir so: Das Setup würde dann, mangels gespeicherter Versionsinformationen, immer nur melden, dass es die installierte Version nicht identifizieren kann.
Zum Speichern der Informationen habe ich eine kleine Hilfsfunktion geschrieben

```
function SaveVersionInfo: boolean;
begin
  Result :=
    (RegWriteDWordValue(HKLM,'{#UNINSTKEY}','MajorVer',
       CreateDWord({#MAJOR},{#MINOR}))) and
    (RegWriteDWordValue(HKLM,'{#UNINSTKEY}','MinorVer',
       CreateDWord({#RELEASE},{#BUILD})));
end;
```

die ich in der Prozedur "`CurStepChanged`", nach der durchgeführten Installation aufrufe

```
procedure CurStepChanged(CurrentStep: TSetupStep);
begin
  if(CurrentStep = ssPostInstall) then
    SaveVersionInfo;
end;
```

Ich kann vorher nämlich nicht auf den Uninstall-Schlüssel in der Registry zugreifen. Das heißt, ich kann schon, nur würden meine Einträge dann wieder gelöscht werden. Das ist ein kleines Problem von Inno Setup, das ich nicht ganz nachvollziehen kann.

Laut Installationslogik werden bspw. relativ kurz vor dem Ende der Installation die Einträge der Sektion "`[Registry]`" geschrieben. Ich könnte also die Anweisungen zum Speichern der Versionsdaten auch dort unterbringen. Danach werden Dateien registriert (sofern nötig), und dann wird der Eintrag für den Uninstaller erzeugt.
Allerdings entfernt Inno Setup einen evtl. vorhandenen Uninstall-Schlüssel restlos und legt ihn komplett neu an. Und hätte ich nun die Versionsinformationen per "`[Registry]`"-Sektion schreiben lassen, wären diese dadurch gelöscht worden.
Deshalb muss ich bis zum Ende der tatsächlichen Installation warten, bevor ich die Informationen schreiben kann. Und dabei hilft mir die o.g. Prozedur mit dem *ssPostInstall*-Flag.

Eine kleine Erweiterung

Wenn Sie wissen möchten, welche Version installiert ist, dann ändern Sie bitte zunächst die Fundmeldung um

```
[CustomMessages]
de.NewerAppFound=Es ist bereits die Version %s von \
  {#APPNAME} installiert.
```

Der Parameter "%s" soll später durch die Nummer der gefundenen, installierten Version ersetzt werden. Weil diese aber in Form von zwei `dwords` vorliegt, muss sie zunächst wieder in einen `string` umgewandelt werden.
Sie erinnern sich noch an das Prinzip (vgl. Seite 49*ff*)? Wir haben jeweils zwei Teile der Nummer in einem `dword` zusammengefasst. So wurde zum Beispiel die Hauptversion um 16 Bit nach links geschoben, und die Nebenversion wurde dann einfach addiert. Die Nebenversion ließe sich nun am einfachsten ermitteln, indem man die oberen 2 Bytes ausblendet. Und das kann man mit Hilfe einer `and not`-Anweisung und der Bitmaske `0xffff0000` tun.
Und die Hauptversion erhält man, indem man den Wert einfach wieder um 16 Bits nach rechts schiebt. Dadurch verschwinden die unteren 2 Bytes. Das hört sich möglicherweise komplizierter an, als es tatsächlich ist. Die ganze Funktion zum Dekodieren der Versionsnummer besteht nur aus diesen paar Zeilen

```
function DecodeVersion(const dwMajor, dwMinor: dword): string;
var
  a,
  b,
  c,
  d       : word;
begin
  a       := word(dwMajor shr 16);
  b       := word(dwMajor and not $ffff0000);

  c       := word(dwMinor shr 16);
  d       := word(dwMinor and not $ffff0000);

  Result := Format('%d.%d.%d.%d',[a,b,c,d])
end;
```

Nun müssen wir nur noch die Dialogbox in der Funktion "`IsSetupNewer`" etwas anpassen. Sie zeigt ja bisher nur die Meldung, dass eine aktuellere Version gefunden wurde. Diese Meldung ist ja bereits angepasst worden, so dass wir den Platzhalter nun ebenfalls mit Hilfe der gerade gezeigten Funktion "`Format`" durch die dekodierte Versionsnummer ersetzen können

```
  if(not Result) then
    MsgBox(Format(ExpandConstant('{cm:NewerAppFound}'),
      [DecodeVersion(SavedMajor,SavedMinor)]),mbError,MB_OK);
```

Parallele Installationen

In diesem Kapitel möchte ich die Möglichkeit einer parallelen Installation vorstellen. Im Normalfall ist es so, dass man nur eine Version einer Anwendung installieren kann, und dass diese dann mit einer neuen Version aktualisiert wird.
Manchmal ist es allerdings auch sinnvoll bzw. sogar erforderlich, dass zwei verschiedene Versionen einer Anwendung parallel zueinander installiert sind und unabhängig voneinander laufen. Dass die Anwendungen dies auch unterstützen müssen, dürfte auf der Hand liegen. Aber das ist ein Thema, das mehr mit der Programmierung und weniger mit dem Setup zu tun hat.

In meinem Fall soll es so sein, dass es nur zwei gleichzeitig installierte Versionen einer Anwendung geben kann und darf. Installiert man eine aktuelle Version, dann soll eine eventuell vorhandene ältere Version auf den neuen Stand gebracht werden. Und eine ältere Version soll man parallel zu einer bereits vorhandenen neuen Version installieren können. Als Grundlage dazu dient mir die Versionskontrolle aus dem entsprechenden Kapitel ab Seite 49.

Zuerst benötigen wir zwei IDs für die Anwendung. Diese geben an, ob es sich um eine reguläre oder um eine parallele Installation handelt. Dann benötigt jede Version (regulär und parallel) auch noch einen eigenen Registryschlüssel für die Deinstallationsinformationen

```
#define MAJOR 1
#define MINOR 0
#define RELEASE 0
#define BUILD 0

#define SIMPLEAPPVER Str(MAJOR) + "." + Str(MINOR)
#define APPVER Str(MAJOR) + "." + Str(MINOR) + "." + \
  Str(RELEASE) + "." + Str(BUILD)

#define COMMON_APPID "SIS_COMMON_A630B32E-1985-47A8-8B1B-3454E514E068"
#define APPID "SIS_" + SIMPLEAPPVER + \
  "_A630B32E-1985-47A8-8B1B-3454E514E068"

#define COMMON_UNINSTKEY \
  "Software\Microsoft\Windows\CurrentVersion\Uninstall\" + \
  COMMON_APPID + "_is1"
#define UNINSTKEY \
  "Software\Microsoft\Windows\CurrentVersion\Uninstall\" + APPID + \
  "_is1"
```

Sie sollten in diesem Beispiel sehen, dass die ID für die parallele Version die Versionsnummer mit einschließt, um sich von der regulären Installation zu unterscheiden.

Die bereits bekannte Funktion "`IsSetupNewer`" aus der Versionskontrolle muss nun so geändert werden, dass sie eine ältere Version akzeptiert. Wie gehabt werden zunächst die gespeicherten Informationen einer bereits installierten Version ermittelt

```
function IsSetupNewer: boolean;
var
  tmp          : string;
begin
  if(not RegQueryDWordValue(HKLM, '{#COMMON_UNINSTKEY}', 'MajorVer',
       SavedMajor)) or
    (not RegQueryDWordValue(HKLM, '{#COMMON_UNINSTKEY}', 'MinorVer',
       SavedMinor)) then
  begin
```

Gibt es keine solchen Informationen, dann ermittelt das aktuelle Setup den Namen und Pfad des Deinstallationsprogramms und setzt dann von sich aus voraus, dass die installierte Version eine aktuellere ist. Dies mag zwar, mangels Versionsinformationen, nicht in jedem Fall stimmen, dürfte aber sicherer und eher im Interesse des Anwenders sein, als die installierte Version einfach zu überschreiben. Ich nutze hier das erste Mal eine neue globale `bool`-Variable "`ParallelMode`"

```
    if(RegQueryStringValue(HKLM, '{#COMMON_UNINSTKEY}',
         'UninstallString', tmp)) and
      (tmp <> '') and
      (fileexists(tmp)) then
    begin
      ParallelMode := true;
      exit;
    end;
  end;
```

Im anderen Fall erzeugt das Setup nun mit seinen eigenen Versionsinformationen die nötigen Daten

```
  SetupMajor    := CreateDWord({#MAJOR},{#MINOR});
  SetupMinor    := CreateDWord({#RELEASE},{#BUILD});
```

und vergleicht diese dann mit den gefundenen und anfangs ermittelten Daten

```
  ParallelMode := (SavedMajor > SetupMajor) or
    ((SavedMajor = SetupMajor) and (SavedMinor > SetupMinor));
```

Sie können sehen, dass die neue bool-Variable auf `TRUE` gesetzt wird, wenn die gespeicherte Version aktueller ist. Die Funktion "`IsSetupNewer`" selbst liefert nun in jedem Fall `TRUE` als Ergebnis zurück

```
  Result       := true;
end;
```

In der Ereignisfunktion "`InitializeSetup`" setzen wir nun alle Variablen quasi auf Null und rufen dann die angepasste Funktion "`IsSetupNewer`" auf

```
function InitializeSetup: boolean;
begin
  SetupMajor    := 0;
  SetupMinor    := 0;
  SavedMajor    := 0;
  SavedMinor    := 0;
  ParallelMode := false;

  Result       := IsSetupNewer;
end;
```

Verzeichnis- und Gruppennamen anpassen

Im nächsten Schritt müssen wir dafür sorgen, dass bei einer parallelen Installation der Zielordner und die Startmenügruppe angepasst werden. Bei der Gelegenheit wird auch gleich die "`AppId`" geändert

```
[Setup]
DefaultDirName={code:GetPatchedDestination|{pf}\SISTestSetup}
DefaultGroupName={code:GetPatchedDestination|Side-by-Side}
AppId={code:GetPatchedAppId|}
```

```
[Code]

function GetPatchedDestination(DefaultValue: string): string;
begin
  if ParallelMode then
    Result := Format('%s.%s', [DefaultValue, '{#SIMPLEAPPVER}'])
  else
    Result := DefaultValue;
end;

function GetPatchedAppId(dummy: string): string;
begin
  if ParallelMode then
    Result := '{#APPID}'
  else
    Result := '{#COMMON_APPID}';
end;
```

Die Erklärung ist recht einfach: Bei einer parallelen Installation benutzt das Setup zwar die Vorgaben, hängt aber zusätzlich noch die aktuelle Versionsnummer an. Aus dem Gruppennamen "Side-by-Side" wird dann eben zum Beispiel "Side-by-Side 1.0".
Und die geänderte "`AppId`" entscheidet über den benutzten Registryschlüssel, so dass sich die beiden installierten Versionen tatsächlich nicht in die Quere kommen.

Den Anwender informieren

Ich habe mich in meinem Beispiel für eine zusätzliche Seite im Setup entschieden, um den Anwender zu informieren. Es wirkt meiner Meinung nach etwas professioneller und bietet zudem auch die Möglichkeit, die schon vorhandene Version deinstallieren zu lassen.
Eigene Seiten in der Setuplogik hatten wir bereits besprochen, so dass ich an der Stelle nur auf die entsprechenden Kapitel ab Seite 30 verweisen möchte. Hier die Kurzfassung

```
var
  ParallelModePage : TinputOptionWizardPage;

procedure InitializeWizard;
begin
  ParallelModePage :=
    CreateInputOptionPage(wpWelcome,
      'Parallele Installation',
      'bla bla bla',
      'Es wurde die Version X gefunden ...',
      true,
      false);
```

In der Beschreibung (fett markiert) könnten Sie zum Beispiel mit Hilfe des "`Format`"-Befehls anzeigen lassen, welche Version das Setup enthält, und welche Version bereits installiert ist.

Der Anwender kann nun die bereits vorhandene aktuelle Version entfernen lassen, oder er installiert das Setup quasi parallel dazu

```
  ParallelModePage.Add('vorhandene Version entfernen');
  ParallelModePage.Add('parallele Installation');
  ParallelModePage.Values[1] := true;
end;
```

Jetzt müssen wir noch zwei Dinge tun: Wenn es sich um eine reguläre Installation handelt, und es gar keine installierte aktuelle Version gibt, dann darf diese neue Seite natürlich nicht zu sehen sein

```
function ShouldSkipPage(PageId: integer): boolean;
begin
  Result := ((PageId = ParallelModePage.ID) and (not ParallelMode)) or
```

Handelt es sich aber um eine parallele Installation, dann sollen die neuen Vorgaben für den Zielordner und die Programmgruppe benutzt werden, die der Anwender nicht ändern darf. Hier werden also Verzeichnis- und Gruppenauswahl ausgeblendet

```
    (((PageId = wpSelectDir) or (PageId = wpSelectProgramGroup)) and
     ParallelMode);
end;
```

Bleibt noch die Frage, wie man die bereits vorhandene Version entfernen kann, ohne dass man dazu das aktuell laufende Setup abbrechen muss. Wir nutzen die Funktion "`NextButtonClick`", die immer ausgelöst wird, wenn man auf den Weiter-Button klickt.
In der Funktion vergleichen wir die übermittelte ID der aktuellen Setupseite mit der ID unserer neu erzeugten Seite. Stimmen beide überein, und ist auch die erste Option zur Deinstallation einer vorhandenen Version aktiv, dann rufen wir die dazu passende Funktion auf.

```
function NextButtonClick(CurrentPageId: integer): boolean;
begin
  if (CurrentPageId = ParallelModePage.ID) and
     (ParallelModePage.Values[0]) then
  begin
    { alte Version entfernen, s. Kapitel auf Seite 35 }
  end;

  Result := true;
end;
```

Versionsinformationen speichern

Es fehlen noch die Versionsinformationen, damit das Setup bei einem erneuten Start selbige auch ermitteln und vergleichen kann. Wir können hier die Originalfunktionen aus dem Kapitel ab Seite 49 nutzen, ändern aber "`SaveVersionInfo`" wie folgt ab, damit die Daten nur gespeichert werden, wenn es sich um eine reguläre Installation handelt

```
function SaveVersionInfo: boolean;
begin
  if ParallelMode then
  begin
    Result := true;
    exit;
  end;

  { ... }
end;
```

Sammelsurium

Installierte Produkte modifizieren

Ab Windows 2000 können Sie im Softwaremodul der Systemsteuerung einen separaten Button zum Anpassen eines bereits installierten Produktes benutzen. Diese Technik ist zwar eigentlich für den Microsoft Installer gedacht, funktioniert aber auch mit Inno Setup.
Dazu müssen Sie entweder den Pfad zum Setup angeben, wenn sich dieses bspw. auf einer CD o.ä. befindet, oder Sie kopieren das Setup während der Installation in den Zielordner. Das sollten Sie aber auch nur machen, wenn die Setupdatei nicht zu groß ist.

Ich möchte Ihnen als Beispiel zeigen, wie das Setup in den Zielordner kopiert wird und dort dann zur Verfügung steht. Dazu ergänzen Sie bitte zunächst den folgenden Eintrag in Ihrer "`[Setup]`"-Sektion

```
[Setup]
AppModifyPath="{app}\YourSetup.exe" /modify=1
```

Der Parameter ist frei wählbar, nur korrekt abfragen müssen Sie ihn. Notwendig ist er, damit das Setup später bemerkt, dass es sozusagen im Wartungs- bzw. Anpassenmodus läuft. Zur korrekten Abfrage benötigen wir eine `bool`-Variable und eine Zeile in der "`InitializeSetup`"-Funktion

```
[Code]
var
  Maintenance : boolean;

function InitializeSetup: boolean;
begin
  // ist der Maintenance-Mode aktiv?
  Maintenance := ExpandConstant('{param:modify|0}') = '1';

  // wie auch immer: Setup starten
  Result      := true;
end;
```

Das Prinzip dürfte klar sein. Wird der Parameter nicht übergeben, dann ist die Variable `FALSE`. Wird er übergeben, und wird festgestellt, dass der Wert des Parameters 1 ist, dann ist die Variable `TRUE`.

Jetzt geht es darum, das Setup in den Zielordner zu kopieren. Da es sich hierbei um eine externe Datei handelt, denn das Setup ist ja nicht in sich selbst komprimiert, benutzen wir das `external`-Flag. Wenn wir uns aber im Wartungsmodus befinden, dann wurde das Setup ja bereits aus dem Zielordner heraus aufgerufen. In dem Fall soll es also nicht noch einmal kopiert werden. Und dabei hilft uns die eben deklarierte `bool`-Variable. Ab Inno Setup 5 können Sie einfach und elegant darauf zugreifen

```
[Files]
Source: "{srcexe}"; DestDir: "{app}"; DestName: "YourSetup.exe"; \
  Flags: external ignoreversion; Check: not Maintenance
```

Noch mal zur Erinnerung: Wird das Setup ohne Parameter gestartet, dann ist die Variable `FALSE`. Damit die Setupdatei nun selbst kopiert wird, muss der "`Check`"-Parameter aber `TRUE` liefern. Aus dem Grund wandeln wir die Variable mit `not` um.

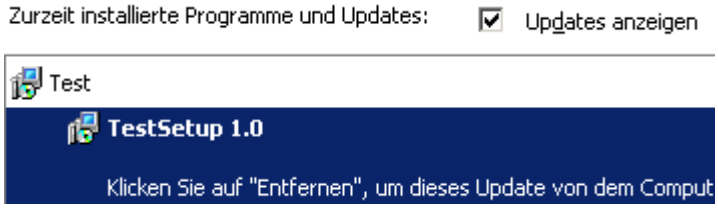
Wenn das Setup nun im Wartungsmodus läuft, dann ist die Auswahl des Zielordners nicht mehr notwendig, denn Ihr Produkt ist ja bereits installiert worden. Das betrifft auch die Angabe der Gruppe im Startmenü und ggf. weitere Abfragen.

Mit Hilfe der Ereignisfunktion "`ShouldSkipPage`" können Sie diese Seiten überspringen. Immer wenn Sie das Ergebnis dieser Funktion auf `TRUE` setzen, wird die jeweils aktuelle Seite übersprungen. Deswegen dürfte der Code auch nicht allzu schwer zu verstehen sein.
Wenn der Wartungsmodus aktiv ist, dann ist die anfangs deklarierte `bool`-Variable `TRUE`. Um nun aber nicht jede Seite abzubrechen, prüfen Sie die ID der Seite, die im Prozedurkopf übergeben wird. *wpSelectDir* stünde zum Beispiel für die Auswahl des Zielordners. *wpSelectProgramGroup* wäre die Seite zur Auswahl der Startmenügruppe

```
function ShouldSkipPage(CurrentPage: integer): boolean;
begin
  Result := Maintenance and
    ((CurrentPage = wpSelectDir) or
     (CurrentPage = wpSelectProgramGroup));
end;
```

Updates unter Windows XP SP2 auflisten

Unter Windows XP SP2 gibt es eine neue Funktion, die für Ordnung in der Liste der installierten Software sorgen kann. Diese Funktion blendet Updates zu einem installierten Produkt nur auf Wunsch des Anwenders ein. Anhand eines Beispielbildes möchte ich Ihnen das zeigen



Um den Eintrag "TestSetup 1.0" sehen zu können, müssen Sie den Haken in der Option oben links setzen. Diese Funktion macht aber auch nur Sinn, wenn Sie zum Beispiel Erweiterungen für ein bereits installiertes Produkt anbieten. Das könnten zusätzliche Sprachdateien, besondere Plugins oder ähnliche Gimmicks sein. Wichtig ist aber, dass sich all diese Dinge auch wieder entfernen lassen, ohne dabei das Hauptprogramm zu beschädigen oder anderweitig zu beeinflussen.

Erreicht wird das durch zwei zusätzliche `string`-Werte im Uninstall-Schlüssel in der Registry. Wie Sie ja bereits aus dem Kapitel über die Versionskontrolle (vgl. Seite 49*ff*) wissen, kümmert sich Inno Setup um diesen Schlüssel. Der Einfachheit halber habe ich folgende Konstanten deklariert

```
const
  UninstallKey =
    'Software\Microsoft\Windows\CurrentVersion\Uninstall\%s';

  ParentKey    = 'InternalTestkey';
  ParentName   = 'Test';
  ThisAppKey   = 'TestSetup_is1';
```

"`ParentKey`" gibt den Namen des Uninstall-Schlüssels des Produktes an, dem Ihr Update untergeordnet werden soll. Dieser Wert muss aber nicht unbedingt auf existierende Schlüssel verweisen. Im Beispiel habe ich einen Wert benutzt, den es vermutlich auf keinem Rechner gibt. Und in dem Fall kommt dann noch zusätzlich "`ParentName`" zum Tragen.

Wenn Sie den Schlüsselnamen eines installierten Produktes verwenden, wird das Update unter diesem Produkt angezeigt. Für separate Updates brauchen Sie aber eine Art Gruppennamen, damit der Anwender diese Updates zuordnen kann. Ein gutes Beispiel dafür sind die Sicherheitspatches und Updates von Windows selbst.

Dann erinnern Sie sich vielleicht daran, dass ich im Kapitel über die Versionskontrolle schrieb, dass Sie nicht zu früh in den Uninstall-Schlüssel Ihrer eigenen Anwendung schreiben dürfen. Das Setup legt diesen Schlüssel erst recht spät an, und wenn er zu diesem Zeitpunkt schon existiert, wird er kurzerhand entfernt, bzw. der vorhandene Inhalt wird gelöscht.
Wie bei der Versionskontrolle können wir also erst nach der durchgeführten Installation in diesen Schlüssel schreiben. Das Schreiben der Daten selbst richtet sich nach den schon angesprochenen Vorgaben. Der Wert von "`ParentKey`" wird immer in die Registry geschrieben. Dann wird geprüft, ob dieser Schlüssel, auf den "`ParentKey`" verweist, überhaupt existiert. Ist das nicht der Fall, schreibt das Setup noch den Wert von "ParentName" in die Registry, damit die Updates dann ihre eigene Gruppe benutzen

```
procedure SaveUpdateInfo;
begin
  RegWriteStringValue(HKLM, Format(UninstallKey, [ThisAppKey]),
    'ParentKeyName', ParentKey);

  if not RegKeyExists(HKLM, Format(UninstallKey, [ParentKey])) then
    RegWriteStringValue(HKLM, Format(UninstallKey, [ThisAppKey]),
      'ParentDisplayName', ParentName);
end;
```

Diese Prozedur wird nun nach der Installation aufgerufen, und speichert somit die gewünschten Daten.

```
procedure CurStepChanged(CurrentStep: TSetupStep);
begin
  if CurrentStep = ssPostInstall then
    SaveUpdateInfo;
end;
```