kleiner kommunikation
max@kleiner.com

1 PascalScript 3.0 von RemObjects

Wie man mit wirksamen Mitteln eine Applikation mit einer Scripting Engine ausstatten kann, zeigt der Einsatz von PascalScript (PS). Man erhält eine flexible und ausbaufähige Technik die bspw. ein Programm zur Laufzeit beim Kunden erweitert, ganz ohne Neukompilieren oder Installation des Systems.

Anhand des mitgelieferten Skripteditor für Ausbildung, Industrie oder Prototyping, welcher Skript und Ausgabebereich auf einen Blick erlaubt, lässt sich die PS Engine im Folgenden erklären.

1.1 Pascal Power zum Nulltarif

Pascal Script ist eine freie und kostenlose Scripting Engine, die den Aufruf der meisten ObjectPascal Sprachkonstrukte zur Laufzeit in einem Delphi Projekt ermöglicht. PS wurde von Carlo Kok komplett in Delphi entwickelt, der mittlerweile eine Zusammenarbeit mit RemObjects [i] eingegangen ist. RemObjects treibt mit Freude und Freunden die Weiterentwicklung von PS voran. Diese Bibliothek, die sich in die Komponentenpalette von Delphi integrieren lässt, ist als Komponentensammlung strukturiert und durchdacht aufgebaut, da ein Helpsystem, ein eigenes Forum (remobjects.public.pascalscript) und zusätzliche Beispiele das Paket nützlich ergänzen.

PS bietet folgende Merkmale:

- Variablen, Konstanten und Standardkonstrukte
- Funktionen, Rekursion und Standardtypen
- Import von Delphi Funktionen und Klassen
- Zuweisung von Scriptfunktionen zu Delphi Ereignissen
- Kompilieren eines Byte Code Files zum späteren Einsatz
- Einfacher Gebrauch der Design Komponenten

Die Installation ist bequem als Package organisiert. Nach dem Herunterladen der komprimierten Datei (Version 3.0.3.45 vom 12.7.04) erhält man eine Verzeichnisstruktur, worin sich auch Pakete und Sourcen für Delphi 6,7 und Kylix 3 befinden. Für Kylix 2 Entwickler habe ich eine eigene inoffizielle Migration mit zugehörigem Paket erstellt (*PascalScript_kylix2.zip*).

Am Besten starten Sie das Abenteuer mit erwähntem Projekt „TestApplication“ oder rufen direkt die Datei *PascalScripter.exe* auf, um einen ersten Eindruck zu erhalten.

PS besteht hauptsächlich aus den drei grossen Units:

- Compiler (uPSCompiler.pas)
- Runtimer (uPSRuntime.pas)
- Scripting Engine (uPSComponent.pas)

Der Compiler mit zugehörigem Parser implementiert einen stackbasierten „Pascal to Byte Code“ der dann durch die Klasse `TPSExec` des Runtimer geladen und ausgeführt wird. Die Design Time Komponente Scripting Engine (Klasse `TPSScript`) in der Unit `uPSComponent` ist ein Komponenten Wrapper für die Units Compiler und Runtimer, welcher direkt das Ausführen und Debuggen von Scripts ermöglicht. Sogar ein Preprocessor ist an Bord. Den Compiler und Runtimer kann man auch voneinander unabhängig in Projekten aktivieren.

Wozu benötigt man überhaupt eine Scripting Engine?

Angenommen sei eine Delphi Applikation, die den Einsatz oder den Unterhalt von Flugzeugen regelt. Das Skript erlaubt die Nutzung eines TPilot Record , über den man alle relevanten Daten an das Skript

übergibt, als da sind: Startzeit, Abflugzeit, Ankunftszeit, Landezeit, Gewicht, Geschwindigkeit, Streckenlänge etc. Das Skript erhält keine Informationen, wessen Daten gerade verarbeitet werden, deshalb sind entsprechende Manipulationsmöglichkeiten ausgeschlossen.

Das Skript wertet dann die übergebenen Daten aus und liefert das Punktergebnis für den Flug zurück. Generell lässt sich ein System mit einem Skript aktualisieren oder steuern, ohne eine Manipulation auf dem Rechner vornehmen zu müssen.

Beim Gebrauch von PS plazieren Sie die Komponenten `TPSScript` auf eine Form oder ein Datenmodul, weisen ein Skript der Eigenschaft zu und rufen anschliessend die Compile und Execute Methoden auf:

```
Begin
  Memo2.Lines.Clear;
  PSScript.Script.Assign(Memo1.Lines);
  Memo2.Lines.Add('Compiling MaxBox');
  //TPSPascalCompiler transforms to bytecode
  if PSScript.Compile then begin
    OutputMessages;
    Memo2.Lines.Add('MaxBox Compiled succesfully');
    if not PSScript.Execute then begin
      Memo1.SelStart := PSScript.ExecErrorPosition;
      Memo2.Lines.Add(PSScript.ExecErrorToString +' at
'+Inttostr(PSScript.ExecErrorProcNo)+'.'+Inttostr(PSScript.ExecErrorByteCodePosition
));   end else Memo2.Lines.Add('RunMax Succesfully executed');
  end else begin
    OutputMessages;
    Memo2.Lines.Add('Compiling failed');
  end
end;
```

Die Engine plaziert bei einem Fehler des Byte Code sogar den Cursor an die richtige Stelle, auch wenn Sie keinen Debugger einsetzen.

Wie erwähnt sind aus Effizienzgründen nicht alle Funktionen aus dem Sprachvorrat registriert. Eine Liste der registrierten Funktionen ist am Anfang der Unit *uPSCompiler* zu finden. Neue Methoden oder Funktionen sind im registrierten Umfang wie folgt aufzunehmen. Bspw. benötigt man die Funktion random(), die sich innerhalb der Engine im Ereignis `onCompile` registrieren lässt:

```
  Sender.AddFunction(@myrandom, 'function random(const a: byte): byte');
```

Der erste Term ist ein Zeiger auf die Funktion und der zweite deklariert den Namen wie er vom Skript aus angesprochen wird. Die Funktion selbst wird gemappt oder umgelenkt:

```
function myrandom(const a: byte): byte;
begin
  result:= Random(a);
end;
```

Das Registrieren von Objekten die sich innerhalb der Applikation befinden, hat einen ähnlichen Vorgang. Dazu existiert bei grösseren Projekten das Tool *PSUnitImporter*, das mit Hilfe eines Unit Parsers Methoden und Eigenschaften registriert. Nebst den Standardfunktionen sind spezielle Komponenten (siehe Abb.) mit vor registriertem Funktionsumfang erhältlich.

// ps_lib.tif

Hervorragend von PS ist auch die Laufzeitunterstützung, da die Scripting Engine nie direkt `Application.ProcessMessages` aufruft, die Applikation kann also blockieren während das Skript weiter läuft! Mit einem Plugin ähnlichen Mechanismus sind eigene Klassen importierbar. Zum Debuggen benötigen Sie die Bibliothek von SynEdit [ii]. Wie man den Compiler und Runtimer separat gebraucht, ist in den Verzeichnissen /Import und /Kylix ersichtlich.

Kombinieren Sie mit beigelegtem Projekt *PascalScript*.exe ihre eigenen Funktionen oder sprechen direkt die Objekte `memo1` oder `memo2` an, um die Flexibilität eines Interpreters erfahren zu können. Natürlich hat diese Flexibilität ihren Preis, die Geschwindigkeit, die trotz des `PSExecuter` gegenüber dem reinen dcc32 Compilat schon mal um den Faktor 1:40 langsamer daherkommt.

Max Kleiner

Beispiel auf der CD-ROM

Aktuelle Sourcen: http://www.remobjects.com

---

[i] http://www.remobjects.com/

[ii] http://synedit.sourceforge.net/