**Skelett-Records**

Übersicht

Skeleton-Datensätze sind eigentlich Datensatzvorlagen, die es Ihnen ermöglichen, dieselben Vorlagen in verschiedenen Situationen zu verwenden. Diejenigen unter Ihnen, die Delphi verwenden, werden mit dem allgemeinen Konzept und der Verwendung von Generics vertraut sein.

Im Vergleich zu herkömmlichen Datensätzen bieten Skeleton-Datensätze eine Reihe zusätzlicher Vorteile: Sie können Funktionen, Prozeduren und Eigenschaften enthalten, die auf Felder des Datensatzes zugreifen können, ohne den Datensatz direkt zu referenzieren.

Die Felder können auch in begrenztem Umfang mit öffentlichen und privaten Schlüsselwörtern geschützt werden. Da der Basic-Compiler diese Schlüsselwörter jedoch nicht versteht, gilt dieser Schutz nur innerhalb der Skelettstrukturen selbst.

**Syntax**

Erkennen, dass ein Datensatz ein Skelett ist

Die Grunddefinition eines Skeletts ist einem herkömmlichen Datensatz sehr ähnlich, und in der Tat kann jeder vorhandene Datensatz leicht in ein Skelett umgewandelt werden, indem der Definition ein Paar spitze Klammern <> hinzugefügt wird.

**type**

tUSART = **record**<>

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..7] of byte;

fError : tUSART\_ERR;

**end**;

Dies ist absolut identisch mit einer klassischen Definition und kann auf die gleiche Weise umgesetzt werden

**var**

Monitor : tUSART<>;

Bislang gibt es keine Vorteile von Skeletten, aber...

Festlegen von Parametern

Innerhalb der spitzen Klammern können wir Parameter angeben. Wir könnten zum Beispiel die Größe des Puffers angeben, etwa so:

**type**

tUSART = **record**< RxBuffSize : byte >

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..RxBuffSize - 1] of byte;

fError : tUSART\_ERR;

**end**;

Das bedeutet, dass wir mehrere verschiedene Datensätze mit verschiedenen Puffergrößen definieren könnten:

**var**

Monitor1 : tUSART< RxBuffSize = 7>;

Monitor2 : tUSART< RxBuffSize = 9>;

Bei der Verwendung herkömmlicher Datensätze würde dies zwei separate Datensatzdefinitionen erfordern, aber mit Skeletten ist nur eine Definition erforderlich. Intern werden zwar immer noch zwei Definitionen benötigt, aber sie werden vom Compiler und nicht vom Programmierer erzeugt.

**Standard-Parameter**

Es ist möglich, eine Voreinstellung für einen Parameter anzugeben, z. B. :

**type**

tUSART = **record**< RxBuffSize : byte = 8>

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..RxBuffSize - 1] of byte;

fError : tUSART\_ERR;

**end**;

In diesem Fall müssen Sie den Parameter nur angeben, wenn Sie einen anderen Wert benötigen.

**var**

Monitor1 : tUSART<>;

Monitor2 : tUSART< RxBuffSize = 9>;

Funktionen und Prozeduren

Es ist möglich, Funktionen und Prozeduren zu einem Skelett hinzuzufügen, was bei herkömmlichen Datensätzen nicht möglich ist:

**type**

tUSART = **record**< RxBuffSize : byte = 8>

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..RxBuffSize - 1] of byte;

fError : tUSART\_ERR;

**Function** Serstat: Boolean;

**end**;

Hier die Definition dieser speziellen Funktion

**Function** tUSART.Serstat: Boolean;

**begin**

**Return**(fRXRP <> fRXWP);

**end**;

Hier gibt es ein paar Dinge zu beachten. Zunächst einmal bedeuten die spitzen Klammern hier „nicht gleich“, nicht „Skelett“. Zweitens wird auf Datensatzfelder (fRXRP und fRXWP) verwiesen, ohne dass genau angegeben wird, auf welche Datensatzinstanz verwiesen wird. Dies ist ein großer Unterschied zur herkömmlichen Programmierung.

**Var-Parameter**

Es ist möglich, Referenzen auf ein anderes Objekt als Parameter zu übergeben. Dies können andere Datensatzinstanzen sein, einschließlich Skelett-Datensatzinstanzen. Diese werden mit dem Schlüsselwort „var“ angegeben.

**type**

tUSART = **record**< **var** UsesUSART : tLL\_USART<> = USART0;

RxBuffSize : byte = 8>

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..RxBuffSize - 1] of byte;

fError : tUSART\_ERR;

**procedure** SerOut(Value : char);

**Function** Serstat: Boolean;

**procedure** Writeln( Value : string[ RxBuffSize ] );

**end**;

Hier gibt es ein paar Dinge zu beachten. Zunächst einmal bedeuten die spitzen Klammern hier „nicht gleich“, nicht „Skelett“. Zweitens wird auf Datensatzfelder (fRXRP und fRXWP) verwiesen, ohne dass genau angegeben wird, auf welche Datensatzinstanz verwiesen wird. Dies ist ein großer Unterschied zur herkömmlichen Programmierung.

**var**

Monitor1 : tUSART<>;

Monitor2 : tUSART< UsesUSART = USART1; RxBuffSize = 9>;

Die Funktionen können sich auf Parameter genauso beziehen wie auf Felder.

**procedure** tUSART.SerOut(Value : char);

**begin**

UsesUSART.DATA\_TXDATAL := Byte(Value);

**While** not UsesUSART.TXCIF do

**endwhile**;

UsesUSART.TXCIF := TRUE;

**end**;

Und natürlich können Funktionen auf andere Funktionen zugreifen.

**procedure** tUSART.Writeln( Value : string[ RxBuffSize ] );

**var**

i : byte;

**begin**

**for** i := 1 to length( Value ) do

SerOut( Value[i] );

**endfor**;

SerOut( #13 );

**end**;

### **Interrupts**

Skeleton Properties und Public/Private-Schlüsselwörter.

Skelett-Eigenschaften sind ein bisschen wie normale Eigenschaften, aber weniger restriktiv.

Eine normale Eigenschaft erlaubt es, eine Funktion und/oder eine Prozedur zu kombinieren, um sie wie eine Variable zu behandeln. In Sketetons sind sie ähnlich, aber natürlich auf Skelettfunktionen und Prozeduren beschränkt, erlauben aber auch den direkten Zugriff auf Felder. Zum Beispiel:

tUSART = **record**< var UsesUSART : tLL\_USART<> = USART0;

RX : interrupt = USART0\_RXC;

RxBuffSize : byte = 8>

**private**

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..RxBuffSize - 1] of byte;

fError : tUSART\_ERR;

**procedure** SetError( Value : tUSART\_ERR );

**public**

**property** Error : tUSART\_ERR

**read** fError

**write** SetError;

…

**end**;

Beachten Sie, dass wir hier das Lesen als Feld und das Schreiben als Prozedur definiert haben. Wir könnten auch entweder den lesenden oder den schreibenden Teil der Definition weglassen. Wir könnten das read-Element auch als Funktion verwenden.

Hier haben wir die Schlüsselwörter public und private eingeführt, aber ihre Verwendung ist begrenzt. So können z. B. tUSART-Funktionen und -Prozeduren nicht auf die privaten Felder von tLL\_USART<> zugreifen, aber allgemeine globale Funktionen können dies im Moment. Verlassen Sie sich jedoch nicht darauf - das kann sich in Zukunft ändern.

Skelton-Eigenschaften können wie Variablen behandelt werden, so dass in diesem Fall z.B.

Monitor1.Error := xxx;

xxx := Monitor1.Error;

Gemeinsame Nutzung von Funktionen und Prozeduren in verschiedenen Datensätzen

Standardmäßig erzeugt jede Instanz eines Skeletts ihre eigene Kopie der verwendeten Funktionen und Prozeduren. Normalerweise ist dies die richtige Vorgehensweise, aber manchmal gibt es mehrere Instanzen desselben Typs, die eine lange Funktion oder Prozedur verwenden. In solchen Fällen wird es zwar schnell, aber ineffizient, d.h. die Codegröße kann sehr schnell ansteigen. In diesen Fällen kann es sinnvoll sein, dass alle Instanzen eine einzige Kopie der Funktion oder Prozedur verwenden. Dies wird durch die Verwendung des Schlüsselworts record vor dem Schlüsselwort function oder procedure erreicht.

tUSART = **record**< var UsesUSART : tLL\_USART<> = USART0;

RX : interrupt = USART0\_RXC;

RxBuffSize : byte = 8>

**private**

fRXRP : byte; // RX Read Ptr

fRXWP : byte; // RX write Ptr

fRXBuffer : array[ 0..RxBuffSize - 1] of byte;

fError : tUSART\_ERR;

**record** **procedure** SetError( Value : tUSART\_ERR );

**public**

**property** Error : tUSART\_ERR

**read** fError

**write** SetError;

…

**end**;

Beachten Sie, dass dies nur in der Datensatzdefinition und nicht im Implementierungsabschnitt angegeben ist. Dies macht die Änderung des Stils so einfach wie möglich.

Denken Sie daran, dass dies ineffizient ist, was die Geschwindigkeit angeht, und für eine einzelne Instanz ist es auch ineffizient für den Code. Erst bei mehreren Instanzen wird es codeeffizient.

Low-Level-Skeletons und -Treiber

Low-Level-Treiber

Eine typische Verwendung von Skeletons ist die benutzerfreundliche Abbildung von IO-Bereichen von Geräten. Diese werden als Low-Level-Treiber bezeichnet. Diese werden in einem speziellen Verzeichnis unter dem Compiler-Verzeichnis gespeichert. Jedes Gerät hat sein eigenes Verzeichnis.

Im obigen Beispiel ist tLL\_USART ein solches Low-Level-Skelett. In verschiedenen Geräten kann es eine unterschiedliche Anzahl solcher Geräte geben, die sich an unterschiedlichen Adressen befinden. Der Benutzer braucht sich darüber jedoch keine Gedanken zu machen, da der Compiler die richtige Version findet.

Die Instanzen (in diesem Fall USART0, USART1 usw.) werden in einer Datei xxx\_REGS.pas definiert, in diesem Fall also USART\_Regs.pas, die sich ebenfalls im Geräteverzeichnis befindet.

Das bedeutet, dass bei einer Änderung der Gerätedefinition in der Hauptdatei automatisch ein anderer Low-Level-Treiber geladen wird.

Wenn Sie z.B. von einem Gerät mit USART1 zu einem ohne USART1 gewechselt haben, wird die Kompilierung fehlschlagen, da USART1 nicht gefunden wird.

Sie können auch speziell für diese Situation vorsorgen, indem Sie {$IFDEF USART1} oder ähnliche bedingte Defines verwenden.

Low-Level-Treiber sind nicht im Lieferumfang des Compilers enthalten, können aber für jedes beliebige Gerät kostenlos von jedem Benutzer, der bereit ist, aktiv an der Entwicklung von Skeletten und Treibern mitzuwirken, oder gegen eine geringe Gebühr von jedem anderen erworben werden.

High-Level-Treiber und Gerätefamilien

Dank dieser Flexibilität können High-Level-Treiber für eine Reihe ähnlicher Geräte gleichzeitig geschrieben werden. Diese Treiber werden im Verzeichnis „family“ gespeichert. Wenn Sie also zum Beispiel einen AVR64EA28 oder einen AVR32EA16 verwenden, wird der High-Level-Treiber im Familienverzeichnis drivers\AVRxEAx gespeichert. (Die Low-Level-Treiber befänden sich in drivers\AVR64EA28 bzw. drivers\AVR32EA16).