Rocket Uniface Library 10.4

Notices

Copyright

Trademarks

Examples

License agreement

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters

77 4th Avenue, Suite 100

Waltham, MA 02451-1468

USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country and Toll-free telephone number

- United States: 1-855-577-4323
- Australia: 1-800-823-405
- Belgium: 0800-266-65
- Canada: 1-855-577-4323
- China: 400-120-9242
- France: 08-05-08-05-62
- Germany: 0800-180-0882
- Italy: 800-878-295
- Japan: 0800-170-5464
- Netherlands: 0-800-022-2961
- New Zealand: 0800-003210
- South Africa: 0-800-980-818
- United Kingdom: 0800-520-0439

Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to www.rocketsoftware.com/support. In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

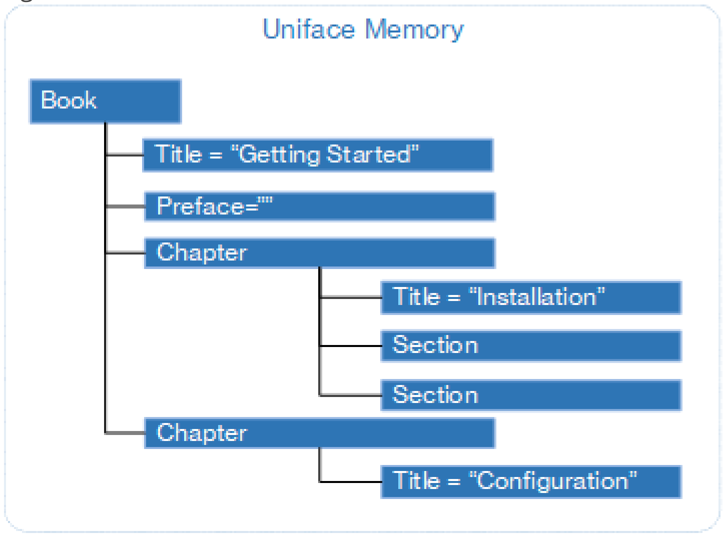
Table of contents

Structs and Members

A Struct is a data structure in memory, consisting of one top node and zero or more sub-nodes called *members*. Each member can hold a scalar data value (such as a number, string, or date), or it can hold another (nested) Struct.

For example, a book is a complex data structure that can be represented by a Struct.

Figure: Struct for a Book



This illustrates a number of characteristics of Structs:

- A Struct consists of a top-level node and an arbitrary number of sub-nodes, known as *members*.

  For example, the Book Struct contains members called `Title`, `Preface`, and `Chapter`.

- A Struct member can have a value. This can be:
  - A scalar value, such as `numeric`, `float`, `string`, `boolean`, `date`, or `time`. For example, the value of first `Title` member is the string `"Getting Started"`.

    A Struct member that has no value, or only a single scalar value, is a *Struct leaf*.

  - A Struct, in which case it is a *nested Struct*. For example, each `Chapter` is a nested Struct because it has child members.
- A Struct can hold a mixture of both scalar members and nested Struct members.

  For example, the Book Struct contains two scalar members (`Title` and `Preface`) and two nested Structs (`Chapter`).

- A Struct or Struct member can have a name.

  If it has a name, the name is case-sensitive. If it is a nested Struct, it is known to its parent by its name. The name can consist of an empty string (`""`), so an expression such as `vStruct->""` refers to all members that have an empty string as a name.

- Members of a Struct can have the same name.

  For example, there are multiple members called `Chapter` and `Section`. Thus, if a `struct` variable references a

Struct by name (for example `vBook->Chapter`), it contains a collection of references to all Structs with that name. For more information, see Struct Dereference Operator (->).

- Struct members are sequentially ordered.

  They can be addressed by their index position, making it possible to address a specific member, even if it has the same name as another member, or no name. For example, `vBook->Chapter{2}` refers to the second chapter member. For more information, see Struct Index Operator ({N}).

- A Struct or nested Struct node can have a special `$tags` Struct containing annotations. This Struct is not counted as a normal member of the Struct. In the example, `$tags` is available (but not shown) on the Book Struct, and on the Chapter Structs. For more information, see Struct Annotations.

Note: The term *Struct node* is often used for the top-level Struct and nested Structs, to distinguish them from scalar Struct members. The term refers to the Struct, excluding child nodes but including its properties, such as `name` and `$tags`.

Most importantly, Structs are dynamic. You can add members to Structs, copy or move members from one Struct to another, and change the values of Struct members. For more information, see Working with Structs.

When working with Structs for XML or XHTML, you may need to add members to Struct leaves, in which case they implicitly become nested Structs with mixed content, or remove members from a nested Structs to make them Struct leaves. For more information, see Struct Leaves.

Related concepts

struct
Transforming Complex Data Using Structs

Struct Leaves

A Struct leaf is a Struct member that has no value, or only a single scalar value. However, adding another value (member) to a Struct leaf implicitly changes it to a nested Struct. The value of the former leaf is now held in a nameless Struct, known as a *scalar Struct*, and the nested Struct has a mixed data structure that may require special handling.

Value of a Struct Leaf

The value of a Struct leaf is returned when you refer to the Struct member. For example, the following statement puts the value of the member `myLeaf` in the message frame:

```
putmess struct1->myLeaf
```

In fact, the value is treated as a member of the leaf, so the function `$membercount` typically returns `1` for a leaf.

However, nested Structs can also have a single member, and sometimes it is important to distinguish between Struct leaves and nested Structs. In such cases, you can use the Struct function `$isLeaf`, which returns `1` for a leaf, and `0` for a non-leaf.

In most cases, this is all you need to know to understand and work with Struct leaves. However, if the Struct embodies a mixed data content model, which combines scalar values with complex values, you need to be able to distinguish between the scalar and non-scalar members. Mixed content is not possible in Uniface component structures, but it is supported by XML and common in XHTML.

Scalar Structs

A scalar Struct is used only to contain the scalar value. It cannot have child nodes, and it is not possible to use access operators (such as `->`) on a scalar Struct. To distinguish a scalar Struct from a Struct node, you can use the `$isScalar` Struct function, which returns `1` for scalar Structs and `0` for Struct nodes.

Important: Most ProcScript does not deal with scalar Structs. For example, a process that loops through the children of a Struct, drilling down to the children that are Structs themselves, should stop where it encounters a Struct leaf. During such loops, test for `$isLeaf` and do not request the members of a leaf. Also `$dbgstring` and `$dbgstringplain` do not display the leaf value (a scalar Struct) as an actual Struct, but merely as a value.

When a Struct leaf is assigned an additional Struct member and becomes a nested Struct, the value remains the same. More precisely, the value of a Struct is the concatenated value of all of its direct members that are scalar Structs. Thus, for the following XHTML code, the scalar value would be `Text can be or`:

```
<div>Text can be <b>bold</b> or <em>italic</em></div>
```

In the following representation, the nameless Struct members are scalar Structs:

```
[]
 [div]
 "Text can be "
 [b] = "bold"
 "or "
 [em] = "italic"
```

You can retrieve all the scalar members of a Struct using the Struct function `$scalar`.

To illustrate the dynamic nature of Struct leaves, consider how to build the following piece of HTML. (part of a table) using Structs.

```
<td>John <b>Smith</b></td>
```

The element `td` has mixed content, containing both text (`John `), and an element (`b`) with only text (`Smith`).

1. First, create a Struct member `td` with value `John `:

   ```
   mystruct->td = "John "
   ```

   At this point, `td`is a leaf, so `mystruct->td->$isLeaf`is `1`

2. Verify this by displaying the Struct in the message frame:

   ```
   putmess mystruct->$dbgstring
   ```

   Because `td`is a leaf, `John`is displayed as a value:

   ```
   []
   ```

```
 [td] = "John "
```

3. Now add a member to represent the bold element containing `Smith`:

   ```
   mystruct->td->*{-1} = $newstruct ; create new member under td
   mystruct->td->*{-1}->$name = "b" ; assign the name
   mystruct->td->b = "Smith" ; assign the value
   putmess mystruct->$dbgstring
   ```

   `td`is now a complex node, so `mystruct->td->$isLeaf`is `0`

   The message frame shows `"John "`as a member, rather than as a mere value:

   ```
   []
    [td]
    "John "
    [b] = "Smith"
   ```

Tip: A more straight-forward way to build this HTML construct is with the following code:

```
mystruct->td = "John " 1
mystruct->td->b = "Smith" 2
```

1 Create a member `td` with value `John`. At this point, `td` is a leaf, so `mystruct->td->$isLeaf` is `1`

2 Add a member `b` to `td`, with value `Smith`. Now `td` is a complex node, so `mystruct->td->$isLeaf` is `0`

Mixed content Structs can also be created by moving one Struct to another. For example:

```
mystruct->td = "John " 1
tmp->b = "Smith" 2
tmp->b->$parent = mystruct->td 3
```

1 Create a member `td` with value `John`. At this point, `td` is a leaf, so `mystruct->td->$isLeaf` is `1`

2 Create another struct with member band value `Smith`. The Struct `tmp` is a placeholder on which a new member named `b` with value `Smith` is created

3 Use `$parent` to move the member `b` to `mystruct`. By default, the member is appended after the last member, so there is no need to set `$index` to `2`.

Effect of XML Attributes on Struct Leaves

Other XML constructs can result in a Struct member being treated as a Struct node instead of a Struct leaf. For example, XML elements can have attributes, which are handled as child elements of their parent element. For example:

```
<div class="note">Text can be bold</div>
```

The `$dbgstringplain` representation looks like this:

```
[]
 [div]
 [class] = note
 Text can be bold
```

The following table shows the values returned by Struct functions for the `div` member, indicating that `div` is a node with 2 members:

Struct Function	Returned Value
`vStruct->div->$isLeaf`	`0`
`vStruct->div->$isScalar`	`0`
`vStruct->div->$memberCount`	`2`

The following table shows the values returned by Struct functions for the nameless scalar member. It shows that it is a scalar Struct with no members:

Struct Function	Returned Value
`vStruct->div->*{2}->$isLeaf`	`1`
`vStruct->div->*{2}->$isScalar`	`1`
`vStruct->div->*{2}->$memberCount`	`0`

Related concepts

- $isLeaf
- $isScalar
- $scalar
- $memberCount

Struct Annotations

Struct annotations are descriptive data elements (also known as *tags*) that are used to correctly interpret data in Structs and convert data to and from Uniface component data, XML, and JSON.

Each Struct node has a special child Struct to hold annotations. This Struct can be accessed using the `$tags` Struct function and is therefore known as the `$tags` Struct.

Unlike normal Struct members, annotations have no specific position inside the Struct, and are therefore not counted or treated as members of the Struct. This is in contrast to, for example, XML processing instructions, which are treated as normal Struct members because their position in an XML document is relevant, even if they are not part of the document contents.

Example: $tags Struct for XML

Consider the following XHTML code:

```
<div class="note">Text can be <b>bold</b></div>
```

When converted to a Struct, each element and attribute is converted into a Struct member, and the `xmlClass` annotation is set to indicate the original XML constructs. This can be clearly seen in the string returned by `$dbgString` Struct function, in which the `$tags` Struct (where present) is always the first child of the Struct member:

```
[]
 [div]
 [$tags]
 [xmlClass] = element
 [class] = note
 [$tags]
 [xmlClass] = attribute
 [] = Text can be
 [b] = bold
 [$tags]
 [xmlClass] = element
```

To get the value of a particular annotation you can use the `$tags` Struct function. For example, the following code assigns the value of the `xmlClass` annotation of the Struct called `div` to a variable:

```
vClass = vStruct->div->$tags->xmlClass
```

For more information, see $tags and Struct Access Operators.

Annotations for Data Conversion

The ProcScript commands for converting data to and from Structs support fixed sets of tags that are specific to the data format. The annotations typically define the data class or object type and additional metadata unique to the data format.

Because annotations always apply to a specific format, they are ignored when converting to other formats and will be lost. For example, XML annotations are ignored when converting from a Struct to a component.

Note: If annotations are supposed to affect the conversion or the data, you need to ensure that the ProcScript manipulates the Structs based upon the annotations.

When you are creating and populating a Struct in preparation for transformation to another format, you can set your own annotations.

- For `componentToStruct` and `structToComponent`, annotations are used to define the type of Uniface object and the reconnect status of occurrences. For more information, see Struct Annotations for Uniface Component Data.
- For `xmlToStruct` and `structToXml`, annotations are used to indicate the type of XML construct or instruction.

For example, the `xmlClass` tag contains information about how the data was stored in the source XML (for example, attribute or element). This is useful when doing a round-trip conversion, so no information is lost. For more information, see Struct Annotations for XML.

- For `jsonToStruct` and `structToJson`, annotations indicate the JSON class and data types. For more information, see Struct Annotations for JSON.

Some data formats have their own metadata, which can be added as annotations. For example, the XML DOCTYPE specification and XML Declaration are required in XML data.

Annotations for Data Interpretation

Some annotations may help to interpret data correctly. For example, when converting from XML to Struct, the XML Schema may specify that the value was originally of type `Duration`, a type that has no equivalent in Uniface. In this case, the value must be passed as a String but the original data type is added as an annotation so that you can apply logic to handle this data type.

Annotations may also provided as read-only information to the developer. Such annotations can be ignored on the conversion back. For example, when using an XML Schema, the xmlDataType tag is added to specify the primitive type of a scalar element. When converted back using the same XML Schema, `xmlDataType` is ignored. (It is not possible to change the XML Data Type by setting this tag as the data type is determined by the XML Schema.)

User-Defined Annotations

You can also add your own annotations if you are creating customized conversion routines. In this case, you should ensure that you adopt a naming policy that prevents tag name conflicts with future versions of Uniface. For example, you could use tag names that begin with an underscore (_).

⚠ **Important:** Uniface guarantees that future tag names will never use the underscore as first character.

Related concepts

Struct Annotations for XML

Struct Annotations for Uniface Component Data

Structs for JSON Data

$tags

$istags

$dbgString

Related tasks

Example: Tags Inheritance