Rocket software

Rocket Uniface Library 10.4

Struct Code Examples

You can copy and paste the code in this topic into a Uniface trigger or operation (such as the *exec* operation of a test component).

Most samples are free of context. Only the example `STRUCT_CONVERSIONS` is a larger example that assumes two (very small) painted entities.

```
; The following ProcScript modules demonstrate the use of Structs.
; After testing the component, display the message frame to view the results.
call STRUCT_BASICS()         ; Creating, modifying and deleting Structs
call MEMBER_NAME_CHARSET()   ; Using reserved names and special characters in member names
call COPY_STRUCT()           ; Copying Struct members/references
call NAME_INHERITANCE()      ; How member names behave during a copy
call TAGS_INHERITANCE()      ; How tags behave during a copy
call MOVE_STRUCT()           ; Moving a Struct
call REMOVE_MEMBER()         ; Detaching a Struct member
call STRUCT_AS_PARAMS        ; Using Structs as parameters
call IDENTICAL_MEMBER_NAMES(); Member names are not necessarily unique
call MEMBER_REFERENCES()     ; Using (old) references after member reassignment
call STRUCT_COLLECTIONS()    ; The Struct as a reference collection
call STRUCT_LOOPS()          ; Looping over a Struct
call REMOVE_STRUCT_LEVEL()   ; Removing a level from a Struct
call INSERT_STRUCT_LEVEL()   ; Inserting an additional level into a Struct
call STRUCT_CONVERSIONS()    ; Converting between XML/Struct/component

; ========================================================================
;                                PrintHeader
; ========================================================================
; This entry is called by other modules to display the entry header in
; the message frame
entry PrintHeader
params
  string pEntryName : IN
endparams

    putmess "==============================="
    putmess "  run of %%pEntryName%%%"
    putmess "==============================="

end ;- entry printHeader

; ========================================================================
;                                STRUCT_BASICS
; ========================================================================
; This entry shows how:
;  * Multiple references can give access to the same data
;  * Struct data is deleted when there are no more struct variables
;    that refer to it (or any of its parents)

entry STRUCT_BASICS
variables
  struct vStruct, vPerson ; Declare struct reference variables
endvariables
```

```
; Display entry header in the message frame:
  call printHeader("STRUCT_BASICS")

; -- Build up a Struct --
  ; Create a Struct:
  vStruct = $newstruct
     ; Note: A Struct is created on the fly when a member
     ;       is assigned to the variable. This statement is
     ;       therefore allowed, but not required:

  ; Create Struct members:
  vStruct->group = $newstruct
  vStruct->group->person = $newstruct
  vStruct->group->person->firstname = "John"
  vStruct->group->person->email = "john@home.com"
     ; Note: Struct members must be created explicitly;
     ;       they are not created on the fly.

  ; Display the Struct in the message frame:
  putmess "Struct referred to by vStruct: "
  putmess vStruct->$dbgstring
     ; Result:
     ;  []
     ;  [group]
     ;    [person]
     ;      [firstname] = "John"
     ;      [email] = "john@home.com"

; -- Update the Struct --
  ; Get a reference to the 'person' node:
  vPerson = vStruct->group->person
  putmess "Struct referred to by vPerson: "
  putmess vPerson->$dbgstring
     ; Result:
     ;  [person]
     ;    [firstname] = "John"
     ;    [email] = "john@home.com"


  ; Update the Struct refered to by vPerson.
  ; This also updates vStruct->group->person:
  vPerson->email = "john.smith@home.com"
  putmess "Struct referred to by vStruct: "
  putmess vStruct->$dbgstring
  putmess "Note: 'email' changed with vPerson is also changed in vStruct. "
     ; Result:
     ;  []
     ;    [group]
     ;      [person]
     ;        [firstname] = "John"
     ;        [email] = "john.smith@home.com"
     ;  Note: 'email' changed with vPerson is also changed in vStruct.

; -- Delete a Struct --
  ; Display the parent of vPerson
  putmess "The parent of vPerson is: [%%(vPerson->$parent->$name)%%%]"
```

```
    ; Result:  The parent of the 'person' member is: [group]

  ; Assign a different value to vStruct, and display the Structs:
  vStruct = $newstruct
  putmess "After assigning a new value to vStruct, vStruct points to empty Struct:"
  putmess vStruct->$dbgstring
  putmess "vPerson is still valid: [%%(vPerson->firstname)%%%]"
  putmess "but the parent of vPerson is now invalid: [%%(vPerson->$parent->$name)%%%]"
  putmess "resulting in a ProcScript error: %%$procerror: %%$item("DESCRIPTION", $procerrorcontex
t)"
  ; Result:
  ;  After assigning a new value to vStruct, vStruct points to empty Struct:
  ;  [] = ""
  ;  vPerson is still valid: [John]
  ;  but the parent of vPerson is now invalid: []
  ;  resulting in a ProcScript error: -84: Not a valid Handle or Struct specified

end ;- entry STRUCT_BASICS

; ========================================================================
;                              MEMBER_NAME_CHARSET
; ========================================================================
; This entry demonstrates that you can include spaces, special characters,
; or reserved words as member names, by enclosing the name in quotation
; marks when assigning members to a Struct.

entry MEMBER_NAME_CHARSET

variables
  struct vStruct
endvariables

  #comment Use DQ for double quotes to improve readability:
  #define DQ %%"%%%
  ; Display entry header in the message frame:
  call printHeader("MEMBER_NAME_CHARSET")

  ; Use a space in a member name:
  vStruct->"a name" = "abc"
  putmess "Member names can include spaces:"
  putmess vStruct->$dbgstring
  putmess $concat(" For Struct [a name], vStruct->&lt;DQ&gt;a name&lt;DQ&gt; returns: ", %\
    vStruct->"a name")
    ; Result:
    ;  Member names can include spaces:
    ;  []
    ;   [a name] = "abc"
    ;
    ;  For Struct [a name], vStruct->"a name" returns: abc

  ; Use names that conflict with, for example, Struct functions
  vStruct->membername = $newstruct
  vStruct->membername->"$name" = "dollarname"
  putmess "Member names can match Struct functions names:"
  putmess vStruct->membername->$dbgstring
  putmess "  membername->$name   = %%(vStruct->membername->$name)%%%"
```

```
  putmess "  membername-><DQ>$name<DQ> = %%(vStruct->membername->"$name")%%%"
  ; Result:
  ;  Member names can match Struct functions names:
  ;  [membername]
  ;    [$name] = "dollarname"
  ;
  ;  membername->$name   = membername
  ;  membername->"$name" = dollarname

end ;- entry MEMBER_NAME_CHARSET

; ========================================================================
;                                COPY_STRUCT
; ========================================================================
; This entry demonstrates what happens when
; copying references to Structs (copy by reference) and
; copying the Struct itself (copy by value).

entry COPY_STRUCT

variables
  struct vStruct1, vStruct2
endvariables
; Display entry header in the message frame:
  call printHeader("COPY_STRUCT")

; Copy by reference
  ; Build a Struct with two members
  vStruct1->a = "AAA"
  vStruct1->b = "BBB"

  ; Copy vStruct1 to vStruct2 (by reference):
  ;  the left side of the assignment is a struct variable
  vStruct2 = vStruct1

  ; Update using vStruct2:
  vStruct2->b = "BBB-updated"
  putmess "Although vStruct2 changed the Struct, vStruct1->b returns the change:"
  putmess "%%(vStruct1->b)%%%"
  ; Result:
  ; Although vStruct2 changed the Struct, vStruct1->b returns the change:
  ; BBB-updated

; Copy by value
  ; Rebuild the Struct from scratch:
  vStruct1= $newstruct
  vStruct2= $newstruct
  vStruct1->a = "AAA"
  vStruct1->b = "BBB"

  ; Copy vStruct1 to vStruct2 (by value):
  ;  the left side of the assignment is a Struct member
  vStruct2->subnode = vStruct1

  ; Update the copied struct
  vStruct2->subnode->b = "BBB-updated"
```

```
  ; Compare the resulting structs:
  putmess "Struct refered to by vStruct1:"
  putmess vStruct1->$dbgstring
  putmess "Struct refered to by vStruct2->subnode:"
  putmess vStruct2->subnode->$dbgstring
  ; Result:
  ;  Struct refered to by vStruct1:
  ;   []
  ;    [a] = "AAA"
  ;    [b] = "BBB"
  ;
  ;  Struct refered to by vStruct2->subnode:
  ;   [subnode]
  ;    [a] = "AAA"
  ;    [b] = "BBB-updated"

end ;- entry COPY_STRUCT

; ========================================================================
;                               NAME_INHERITANCE
; ========================================================================
; This entry demonstrates that when copying a Struct member,
; the name of the target member is determined by the name
; specified on the left side, if available, and otherwise by
; the name of the right-hand side.

entry NAME_INHERITANCE
variables
  struct vStruct1, vStruct2
endvariables

; Display entry header in the message frame:
  call printHeader("NAME_INHERITANCE")

; Inheritance of member names when left side of assignment has name
  vStruct1 = $newstruct
  vStruct2 = $newstruct
  vStruct1->R = "AAA"
  ; Copy one Struct to the other:
  vStruct2->L = vStruct1->R
  putmess "Left side of assignment has name, so it is used."
  putmess "Name of copied member is 'L':"
  putmess vStruct2->$dbgstring
  ; Result:
  ;  Left side of assignment has name, so it is used.
  ;  Name of copied member is 'L':
  ;  []
  ;   [L] = "AAA"

; Inheritance of member names when left side of assignment has NO name
  vStruct1 = $newstruct
  vStruct2 = $newstruct
  vStruct1->R = "AAA"
  ; Copy one Struct to the other:
  vStruct2->*{-1} = vStruct1->R
```

```
  putmess "Left side of assignment has no name, so name of copied Struct it is used."
  putmess "Name of member is 'R':"
  putmess vStruct2->$dbgstring
  ; Result:
  ;  Left side of assignment has no name, so name of copied Struct it is used.
  ;  Name of copied member is 'R':
  ;  []
  ;    [R] = "AAA"

; Inheritance of member names neither side of assignment specifies a name
  vStruct1 = $newstruct
  vStruct2 = $newstruct
  vStruct1->R = "AAA"
  vStruct2->L = "BBB"
  vStruct2->*{1} = vStruct1->*{1} ;- overwrites member L
  putmess "Neither side of assignment specifies a name, so name of right-hand Struct is used."
  putmess "Name of copied member is 'R':"
  putmess vStruct2->$dbgstring
  ; Result:
  ;  Neither side of assignment specifies a name, so name of right-hand
  ;  Struct is used.
  ;  Name of copied member is 'R':
  ;  []
  ;    [R] = "AAA"

end ; entry NAME_INHERITANCE

; ========================================================================
;                              TAGS_INHERITANCE
; ========================================================================
; This entry demonstrates that annotations (tags) are also copied
; when copying a Struct. However, when assigning a scalar value to
; a Struct, source tags are not affected.

entry TAGS_INHERITANCE
variables
  struct vStruct1
endvariables

; Display entry header in the message frame:
  call printHeader("TAGS_INHERITANCE")


; Tags inheritance when copying a Struct:
  ; Build a Struct and assign annotations (tags) tag (annotation):
  vStruct1->member1 = "value A"
  vStruct1->member1->$tags->someTags = "tag value A"
  vStruct1->member2 = "value B"
  vStruct1->member2->$tags->someTags = "tag value B"
  putmess "Struct with annotations: "
  putmess vStruct1->$dbgstring
  ; Result:
  ;  Struct with annotations:
  ;  []
  ;   [member1] = "value A"
  ;     [$tags]
```

```
  ;       [someTags] = "tag value A"
  ;   [member2] = "value B"
  ;     [$tags]
  ;       [someTags] = "tag value B"

  ; Overwrite member2 with a copy of member1;
  vStruct1->member2 = vStruct1->member1
  putmess "The original tags are replaced when a new Struct is assigned:"
  putmess vStruct1->member2->$dbgstring
  ; Result:
  ;  The original tags are replaced when a new Struct is assigned:
  ;  [member2] = "value A"
  ;    [$tags]
  ;      [someTags] = "tag value A"

; Tags inheritance when assigning a scalar value
  ; Assign a new value to member2:
  vStruct1->member2 = "updated value"
  putmess "The tags are not affected when a scalar value is assigned:"
  putmess vStruct1->member2->$dbgstring
  ; Result:
  ;  The tags are not affected when a scalar value is assigned:
  ;  [member2] = "updated value"
  ;    [$tags]
  ;      [someTags] = "tag value A"

end ;- entry TAGS_INHERITANCE

; ========================================================================
;                           MOVE_STRUCT
; ========================================================================
; This entry demonstrates how an existing struct can be
; inserted as a member of another struct.
entry MOVE_STRUCT
variables
    struct vStruct1, vStruct2, vSteadyRef
endvariables

; Display entry header in the message frame:
  call printHeader("MOVE_STRUCT")

  ; Build Struct vStruct1
  vStruct1->a = "AAA"
  vStruct1->c = "CCC"

  ; Build Struct vStruct2
  vStruct2->b = "BBB"

  ; Make vStruct2->b a member of vStruct1, by updating its $parent:
  vStruct2->b->$parent = vStruct1
  putmess "Struct vStruct2->b moved to vStruct1, and appended as member 'b': "
  putmess vStruct1->$dbgstring
  ; Result:
  ;   Struct vStruct2-&gt;b moved to vStruct1, and appended as member 'b':
  ;   []
  ;     [a] = "AAA"
```

```
  ;     [c] = "CCC"
  ;     [b] = "BBB"

  ; Reposition the newly-added member b (assuming this is relevant)
  ; NOTE: It is no longer possible to use vStruct2->b, because
  ;       it has moved the Struct from vStruct2 to vStruct1.
  ;       For a more elegant approach, see SUGGESTION below.
  vStruct1->b->$index = 2
  putmess "Newly-added Struct 'b' has been repositioned:"
  putmess vStruct1->$dbgstring
  ; Result:
  ;   Newly-added Struct 'b' has been repositioned:
  ;   []
  ;     [a] = "AAA"
  ;     [b] = "BBB"
  ;     [c] = "CCC"

  ;- SUGGESTION : use a dedicated reference for manipulation

  ;-   To prevent the awkward switch from vStruct2 to vStruct1 in these lines:
  ; vStruct2->b = "BBB"
  ; vStruct2->b->$parent = vStruct1
  ; vStruct1->b->$index = vStruct1

  ;- it is easier to use a separate reference for the struct manipulation:
  vStruct2->b = "BBB"
  vSteadyRef = vStruct2->b        ;- vSteadyRef refers the b-struct
  vSteadyRef->$parent = vStruct1
  vSteadyRef->$index = 2

  ;- the end result is the same: vStruct1 is assigned the member 'b' at position 2

end ;- entry MOVE_STRUCT

; ========================================================================
;                           REMOVE_MEMBER
; ========================================================================
; This entry demonstrates how Struct members can be
; removed (detached) from their parent.
; Members are *deleted* only when there are no more
; references to the Struct.

entry REMOVE_MEMBER
variables
  struct vStruct
endvariables

; Display entry header in the message frame
  call printHeader("REMOVE_MEMBER")

  ; Build a Struct with two members
  vStruct->a = "AAA"
  vStruct->b = "BBB"
  putmess vStruct->$dbgstring
  ; Result:
  ;  []
```

```
  ;    [a] = "AAA"
  ;    [b] = "BBB"

  ; Detach b from its parent:
  vStruct->b->$parent = ""
  putmess "Member 'b' has been removed:"
  putmess vStruct->$dbgstring
  ; Result:
  ;  []
  ;    [a] = "AAA"

end ; entry REMOVE_MEMBER

; ========================================================================
;                           STRUCT_AS_PARAMS
; ========================================================================
; This sentry shows the use of Structs as IN, OUT or INOUT parameters
; in entries and partner operations.

; When variables or parameters of type Struct are passed between
; entries or partner operations, a reference to that variable is
; passed. The size of a Struct does not matter because only the
; reference is passed, not a copy of the whole Struct.
; By way of contrast, when a string parameter is passed, a copy of
; the variable is created, which can be expensive if the string
; is very large.

; This example consists of four entries, that demonstrate how
; references to a Struct created in one entry are passed to
; another entry using IN, OUT or INOUT parameters.


; ===========  STRUCT_PARAMS_IN_DO ===========
; This entry sets up the example, performs the calls to the
; other entries, and examines the result of the calls.

entry STRUCT_AS_PARAMS
variables
  struct vStructAsIn
  struct vStructAsOut
  struct vStructPitFall
endvariables

; Display entry header in the message frame
  call printHeader("STRUCT_AS_PARAMS")

; Create the Struct
  vStructAsIn = $newstruct

; The Struct variable vStructAsIn is created and passed to the
; entry STRUCT_PARAMS_IN_DO() as an IN parameter.
; Entry STRUCT_PARAMS_IN_DO manipulates the Struct and returns
; without passing anything back. Both entries reference the same
; Struct, so the calling entry can examine the result of the
; second entry.
```

```
  putmess "(Empty) Struct before call to entry STRUCT_PARAMS_IN_DO:"
  putmess vStructAsIn->$dbgstring
  ; Result:
  ;  (Empty) Struct before call to entry STRUCT_PARAMS_IN_DO:
  ;  [] = ""

; Pass the Struct to entry STRUCT_PARAMS_IN_DO
  call STRUCT_PARAMS_IN_DO(vStructAsIn)

  ; Examine the result
  putmess "vStructAsIn after entry call:"
  putmess vStructAsIn->$dbgstring
  ; Result:
  ;  vStructAsIn after entry call:
  ;  [] = ""
  ;   [aValue] = "1111"

; In STRUCT_PARAMS_OUT_DO, a Struct is created and
; passed back as a reference, pointing to the Struct
; it just created.

  ; Display the Struct referenced by vStructAsOut
  putmess "(Uninitialized, unassigned) Struct before entry call:"
  putmess "  vStructAsOut->$dbgstring returns an empty string: %%(vStructAsOut->$dbgstring)%%%"
  ; Result:
  ;  (Uninitialized, unassigned) Struct before entry call:
  ;    vStructAsOut->$dbgstring returns an empty string:

  call STRUCT_PARAMS_OUT_DO(vStructAsOut)

  ; Examine the Struct
  putmess "vStructAsOut after entry call:"
  putmess vStructAsOut->$dbgstring
  ; Result:
  ;  vStructAsOut after entry call:
  ;   []
  ;     [aValue] = "2222"

  ; When passing a Struct as parameter, the Struct must be created
  ; by the module that passes the Struct. The following call shows
  ; what happens if the Struct is not created prior to passing a
  ; reference to that Struct.

  ; In the call to STRUCT_PARAMS_PITFALL_DO thestruct variable
  ; vStructPitFall is not initialized before calling entry
  ; STRUCT_PARAMS_PITFALL_DO
  ; This sample provides two solutions in the comments:
  ;   Solution #1 is in the calling entry.
  ;   Solution #2 is in the called entry.

  ; Solution #1: Explicitly create a Struct here
  ; vStructPitFall = $newstruct

  ; Problem: Call an entry with struct vStructPitFall IN parameter
  ;          before creating a Struct
```

```
  call STRUCT_PARAMS_PITFALL_DO(vStructPitFall)

; Examine the result
  putmess "Struct manipulations done by the called module are not visible"
  putmess "if the Struct is not created in the calling module."
  putmess "  vStructPitFall->$dbgstring:  %%(vStructPitFall->$dbgstring)%%%"
  ; Result:
  ;  Struct manipulations done by the called module are not visible
  ;  if the Struct is not created in the calling module.
  ;   vStructPitFall->$dbgstring:

  ; Result if solution #1 or #2 has been applied:
  ;  Struct manipulations done by the called module are not visible
  ;  if the Struct is not created in the calling module.
  ;   vStructPitFall->$dbgstring:
  ;    []
  ;     [aValue] = "3333"

end ;- entry STRUCT_AS_PARAMS

; ===========  STRUCT_PARAMS_IN_DO ===========
; This entry takes a Struct as IN parameter and sets its value

entry STRUCT_PARAMS_IN_DO
params
  struct pStruct: IN
endparams

  pStruct->aValue = "1111"

end ;- entry STRUCT_PARAMS_IN_DO

; ===========  STRUCT_PARAMS_OUT_DO ===========
; This entry creates a Struct passes it back as an OUT parameter

entry STRUCT_PARAMS_OUT_DO
params
  struct pStruct: OUT
endparams

; Create a Struct with one member.
; Note: The next instruction is redundant; the Struct is implicitly created in
; the subsequent statement
  pStruct = $newstruct
  pStruct->aValue = "2222"

end ;- entry STRUCT_PARAMS_OUT_DO

; ===========  STRUCT_PARAMS_PITFALL_DO ===========
; This entry demonstrates how to avoid the pitfall of
; forgetting to create a Struct before passing a reference to that
; Struct to another entry.

entry STRUCT_PARAMS_PITFALL_DO
params
  struct pStruct : IN
```

```
  ; Solution #2: Use INOUT parameters
  ; struct pStruct : INOUT
endparams

   ; Add a member to the Struct (which is implicitly created if it doesn't exist)
  pStruct->aValue = "3333"

end ;- entry STRUCT_PARAMS_PITFALL_DO

; ========================================================================
;                         IDENTICAL_MEMBER_NAMES
; ========================================================================
entry IDENTICAL_MEMBER_NAMES
; This entry demonstrates that a Struct may contain multiple
; members with the same name. This makes it possible for it to
; reflect XML documents, which allow multiple elements with the same name.

variables
  struct vStruct
  numeric I
endvariables
; Display entry header in the message frame
  call printHeader("IDENTICAL_MEMBER_NAMES")

  ; Build a Struct with 3 members named 'a'
  vStruct->a = "A1"
  vStruct->a{2} = "A2"    ; Update/add item at position 2
  vStruct->a{-1} = "A3"   ; Add item at the end

  ; vStruct->a returns a collection of 3 references:
  putmess "$collSize of vStruct->a is %%(vStruct->a->$collSize)%%%"
  putmess vStruct->$dbgstring
  ; Result:
  ;  $collSize of vStruct->a is 3
  ;  []
  ;    [a] = "A1"
  ;    [a] = "A2"
  ;    [a] = "A3"

  ; For more information on dealing with collections,
  ; see the example STRUCT_COLLECTIONS()

end ;- entry IDENTICAL_MEMBER_NAMES

; ========================================================================
;                          MEMBER_REFERENCES
; ========================================================================
entry MEMBER_REFERENCES
; This example demonstrates how you can maintain a reference to
; original Struct members, even when they have been assigned a new value.

variables
  struct vStruct1, vStruct2
  string string1
endvariables
; Display entry header in the message frame
```

```
  call printHeader("MEMBER_REFERENCES")

; Multiple members
  ; Build a Struct in which multiple members have the same name,
  ; but different values:
  vStruct1->a = "A1"
  vStruct1->a{2} = "A2"
  vStruct1->a{3} = "A3"

  ; Save a reference to the collection of these members:
  vStruct2 = vStruct1->a   ; Reference to members 'a'

  ; Update the complete member set:
  vStruct1->a = "A-updated"
  putmess "Updated vStruct1: "
  putmess vStruct1->$dbgString
  ; Result:
  ;  []
  ;    [a] = "A-updated"

  ; However, vStruct2 still points to the original collection of 3 members:
  putmess "vStruct2 has not been updated:"
  putmess  vStruct2->$dbgstring
  ; Result:
  ;   [a] = "A1"
  ;   [a] = "A2"
  ;   [a] = "A3"

  ; Note: By using a reference to a collection, you can continue
  ;       to access the original data. In example STRUCT_COLLECTIONS
  ;       this reference technique is used to restore a Struct to
  ;       its previous state.

; Single member nodes
  ; Build a Struct with one member:
  vStruct1->a = "A"

  ; Save a reference to the member
  vStruct2    = vStruct1->a

  ; Update the original Struct member
  vStruct1->a = "A-updated"

  ; This actually creates a new member that overwrites the existing member.
  ; (The new member get the tags the original member had; see TAGS_INHERITANCE)

  putmess "vStruct1 has been updated:"
  putmess  vStruct1->a->$dbgstring
  putmess "vStruct2 points to the original member:"
  putmess  vStruct2->$dbgstring
  ; Result:
  ; vStruct1->a has been updated:
  ; [a] = "A-updated"
  ;
  ; vStruct2 points to the original member:
  ; [a] = "A"
```

```
end ;- entry MEMBER_REFERENCES
; ========================================================================
;                           STRUCT_COLLECTIONS
; ========================================================================
entry STRUCT_COLLECTIONS
; This entry demonstrates that a struct variable is always
; a reference to a *collection* of structs, and shows how
; you can work with collections.
; Note: A collection of one looks and works like a
;       reference to a single Struct, but it remains a
;       collection for which $collSize returns 1.
variables
  struct vStruct1, vStruct2, vSomeStructs, vOriginalCollection
  numeric I
  string vDescr
endvariables
; Display entry header in the message frame
  call printHeader("STRUCT_COLLECTIONS")

  ; Prepare two structs
  vStruct1->$name = "Struct1"
  vStruct2->$name = "Struct2"

  ; Assign a member to the collection 'a',
  ; overwriting any existing members named 'a'
  vStruct1->a    = "A1"
  vStruct1->a{2} = "A2"  ; Add a member at position 2
  vStruct1->a{-1} = "A3" ; Append new member at end

  vStruct1->b    = "B"
  putmess vStruct1->$dbgstring
  ; Result:
  ;  [Struct1]
  ;    [a] = "A1"
  ;    [a] = "A2"
  ;    [a] = "A3"
  ;    [b] = "B"

  ; A struct variable is a reference to any number of Structs.
  ; Thus this statement: structVar = aStruct->*
  ; assigns the references to all children of 'aStruct'
  ; to structVar.

  ; The collection size of vStruct1->a (=3) and vStruct1->* (=4 all children)
  putmess "The collection size of vStruct1->a is %%(vStruct1->a->$collsize)%%%"
  ; Use the collection operator ->* to get all references to all children of a Struct
  putmess "The collection size of vStruct1->* is %%(vStruct1->*->$collsize)%%%"
  ; Result:  The collection size of vStruct1->a is 3
  ;          The collection size of vStruct1->* is 4

  ; Looping over the collection vStruct1->a:
  putmess "%%^Loop over all members of vStruct1->a:"
  I = 1
  while (I <= vStruct1->a->$collSize)
```

```
   putmess "  vStruct1->a{%%i%%} has value %%(vStruct1->a{I})%%%"
   I = I + 1
 endwhile
 ; Result:
 ; Loop over all members of vStruct1->a:
 ;   vStruct1->a{1} has value A1
 ;   vStruct1->a{2} has value A2
 ;   vStruct1->a{3} has value A3

; Before reassigning Structs, save a reference to the original
 vOriginalCollection = vStruct1->a ; Keep reference to original collection

 ; Update one specific member:
 vStruct1->a{1} = "A1 - updated"
 putmess "%%^Updated vStruct1->a{%%1%%}: %%(vStruct1->a{1})%%%"
 ; Result:  Updated vStruct1->a{1}: A1 - updated

 ; A collection of multiple Structs has no value:
 putmess "%%^A collection of multiple Structs always returns an empty value"
 putmess " vStruct1->a = %%(vStruct1->a)%%%"
 ; Result:  A collection of multiple Structs always returns an empty value
 ;               vStruct1->a =

 ; A collection with only one Struct behaves like a single Struct:
 vStruct1->a = "A1 - collection reassigned"
 putmess "However, if the collection contains a single Struct,"
 putmess " it is treated as a single Struct:"
 putmess "   vStruct1->a    = %%(vStruct1->a)%%%"
 putmess "   vStruct1->a{1} = %%(vStruct1->a{1})%%%"
 ; Result:
 ;  However, if the collection contains a single Struct,
 ;   it is treated as a single Struct:
 ;     vStruct1->a    = A1 - collection reassigned
 ;     vStruct1->a{1} = A1 - collection reassigned

 ; Restore the original collection
 vStruct1->a = vOriginalCollection

 ; -------------------
 ;  Struct functions and assignments on vStruct1->a can take place
 ;  on all Structs in a collection. $dbgstring is a good example:
 ;  vStruct1->*->$dbgstring prints the collection of all children.
 ;
 ;  Some other struct functions are meaningfull only under conditions:
 ;
 putmess "%%^Using Struct functions on collections:"
 putmess "  All Structs vStruct1->a share the same name: [%%(vStruct1->a->$name)%%%]"
 reset $procerror
 putmess "    but not all vStruct1->*                  : [%%(vStruct1->*->$name)%%%]"
 vDescr = $item("DESCRIPTION", $procerrorcontext)
 putmess "    ProcScript error: %%$procerror%%%: %%vDescr%%%"
 putmess "  All Structs vStruct1->* share their parent : [%%(vStruct1->*->$parent->$name)]"
 vSomeStructs{1} = vStruct1->*
 vSomeStructs{-1} = vStruct1
 reset $procerror
 putmess "    but not all Structs in any collection    : [%%(vSomeStructs->$parent)]"
```

```
vDescr = $item("DESCRIPTION", $procerrorcontext)
putmess "    ProcScript error: %%$procerror%%%: %%vDescr%%%"
; Result:
;  Using struct functions on collections:
;  All structs vStruct1->a share the same name: [a]
;    but not all vStruct1->*->$name           : []
;    ProcScript error: -1151: Structs do not have a common name or parent
;  All structs vStruct1->* share their parent : [Struct1]
;    but not all structs in any collection    : []
;    ProcScript error: -1151: Structs do not have a common name or parent


;  Manipulating collections using Struct functions:
;   one line of code to move all members a from vStruct1 to vStruct2:
putmess "%%^Manipulating structs in collections: Assign $parent:"
vStruct1->a->$parent = vStruct2
putmess "  The members 'a' have moved to vStruct2:"
putmess vStruct1->$dbgstring
putmess vStruct2->$dbgstring
; Result:
;  Manipulating structs in collections, assign $parent:
;    The members 'a' have moved to vStruct2:
;  [vStruct1]
;    [b] = "B"
;
;  [vStruct2]
;    [a] = "A1 - updated"
;    [a] = "A2"
;    [a] = "A3"


putmess "%%^Manipulating Structs in collections: Assign $name:"
putmess "  Members 'a' renamed to 'AAA':"
vStruct2->a->$name = "AAA"
putmess vStruct2->$dbgstring
; Result:
;  Manipulating Structs in collections: Assign $name:
;    Members 'a' renamed to 'AAA':
;  [vStruct2]
;    [AAA] = "A1 - updated"
;    [AAA] = "A2"
;    [AAA] = "A3"
; -------------------
; Assign a subnode to each 'a' member:
vStruct2->*->x = "xyz"
putmess "%%^Member 'x' assigned to all Structs in a collection:"
putmess vStruct2->$dbgstring
; Result:
;  Member 'x' assigned to all Structs in a collection:
;  [vStruct2]
;    [AAA]
;      "A1 - updated"
;      [x] = "xyz"
;    [AAA]
;      "A2"
;      [x] = "xyz"
```

```
  ;    [AAA]
  ;      "A3"
  ;      [x] = "xyz"

end ;- entry STRUCT_COLLECTIONS

; ============================================================================
;                            STRUCT_LOOPS
; ============================================================================

entry STRUCT_LOOPS
; This example shows how to loop over a Struct to perform actions
; on the various levels. The example calls two other entries:
;  REMOVE_TAGS  simple loop that removes the tags on all (sub)structs
;  PRINT_STRUCT slightly more complex loop that prints a
;               representation of the Struct to the message frame
;               that is similar to $dbgstring

variables
  struct vStruct
endvariables
; Display entry header in the message frame
  call printHeader("STRUCT_LOOPS")

  ; Create a Struct:
  vStruct ->$name = "Demo Struct"
  vStruct ->$tags->purpose = "sample"
  vStruct ->description = "Sample Struct"
  vStruct ->subNode = $newstruct
  vStruct ->subNode->$tags->purpose = "sample too"
  vStruct ->subNode->a = "AAA"
  vStruct ->subNode->b = "BBB"

  ; Print the Struct
  putmess "Print the whole Struct:"
  call PRINT_STRUCT(vStruct , "    ")
  ; Result
  ;  Print the whole Struct:
  ;   [Demo Struct]
  ;     [$tags]
  ;       [purpose] = "sample"
  ;     [description] = "Sample Struct"
  ;     [subNode]
  ;       [$tags]
  ;         [purpose] = "sample too"
  ;       [a] = "AAA"
  ;       [b] = "BBB"

  ; Remove the tags from all levels of the Struct
  call REMOVE_TAGS(vStruct )
  putmess "%%^After removing tags from the Struct:"

  ; Pass a collection of children :
  putmess "  Print a collection of the children of the Struct:"
  call PRINT_STRUCT(vStruct ->*, "    ")
  ; Result
```

```
  ;  After removing tags from the Struct:
  ;   Print a collection of the children of the Struct:
  ;    [description] = "Sample Struct"
  ;    [subNode]
  ;      [a] = "AAA"
  ;      [b] = "BBB"

end ; entry STRUCT_LOOPS


; ========================================================================
;                          REMOVE_TAGS
; ========================================================================

entry REMOVE_TAGS
; This entry loops over a Struct collection, recursively stripping
; tags from the Struct and all its children.
; The entry body strips tags from the Struct that was passed in, but
; not from its children. To deal with the child level, the entry
; calls itself recursively.

;  When a collection is passed in, the entry calls itself for each
;  Struct in the collection. The action is applied to individual
;  Structs only (or, more precise: on collections with $collsize = 1)

params
  struct pStruct: in  ; A reference can refer to one or more Structs
endparams
variables
  numeric I, N
endvariables

  if (pStruct->$collSize > 1) ; Multiple Structs in the collection
    I = 1
    N = pStruct->$collSize
    while (I <= N)
      ; Call this entry recursively for each Struct in the collection
      call REMOVE_TAGS(pStruct{I})
      I = I + 1
    endwhile
  else  ; Single Struct, so strip tags:

    pStruct->$tags->*->$parent = ""

    ; Recursively call this entry for all children of the current Struct.
    ; This is a single call, passing all children to the next level:
    if (pStruct->$membercount > 0) call REMOVE_TAGS(pStruct->*)
 endif

end ;- entry REMOVE_TAGS
; ========================================================================
;                          PRINT_STRUCT
; ========================================================================

entry PRINT_STRUCT
; The structure of this entry is similar to REMOVE_TAGS,
```

```
; but  it prints a representation of the Struct to the message frame
; that is similar to $dbgstring.

; Note: $dbgstring is slightly richer in functionality and more
;       efficient than this ProcScript implementation, but this example
;       shows how you can write a custom print routine for Structs.
;       The output of $dbgstring may change over Uniface versions,
;       but a custom  implementation enables you to determine
;       a specific format yourself.

params
  struct pStruct: in  ; A reference can refer one or more structs
  string pMargin: in  ; Initial margin + indenting for deeper levels
endparams
variables
  numeric I, N
endvariables

  #comment using a define for double quotes to improve readability:
  #define DQ %%"%%%

  if (pStruct->$collSize > 1) ; Multiple structs in the collection
    I = 1
    N = pStruct->$collSize
    while (I <= N)
      ; Call this entry recursively for each Struct in the collection
      call PRINT_STRUCT(pStruct{I}, pMargin)
      I = I + 1
    endwhile
  else  ; Single struct, so start printing:
    if (pStruct->$isTags)
      putmess $concat(pMargin, "[$tags]", pStruct)  ; Use label '$tags'
    elseif (pStruct->$isLeaf)
      if (pStruct->$isScalar)
        ; Leaf is already printed on preceding level:
        if (!pStruct->$parent->$isLeaf)
          ; Scalar, so print: value
          putmess $concat(pMargin, "<DQ>%%pStruct%%%<DQ>")
        endif
      else
        ; Leaf, so print: [name] = value
        putmess $concat(  %\
            pMargin, "[", pStruct->$name, "] = <DQ>%%pStruct%%%<DQ>")
      endif
    else
      ; Complex Struct, so print: [name]
      putmess $concat(pMargin, "[", pStruct->$name, "]")
    endif

    ; Print tags, if applicable:
    if (pStruct->$tags->$memberCount > 0)
      call PRINT_STRUCT(pStruct->$tags, $concat(pMargin, "  "))
    endif
    ; End printing
```

```
    ; Recursively call this entry for all children of the current Struct.
    ; This is a single call, passing all children to the next level:
    if (pStruct->$membercount > 0)
      call PRINT_STRUCT(pStruct->*, $concat(pMargin, "  "))
    endif
  endif

end ;- entry PRINT_STRUCT

; ========================================================================
;                           REMOVE_STRUCT_LEVEL
; ========================================================================
; This entry shows how to remove a level from a Struct
; (the reverse of the INSERT_STRUCT_LEVEL)

entry REMOVE_STRUCT_LEVEL
variables
  struct vStruct
endvariables
; Display entry header in the message frame
  call printHeader("REMOVE_STRUCT_LEVEL")

  ; Build a struct
  vStruct->levelA = $newstruct
  vStruct->levelA->levelB = $newstruct
  vStruct->levelA->levelB->childX = "xx"
  vStruct->levelA->levelB->childY = "yy"
  vStruct->levelA->levelB->childComplex = $newstruct
  vStruct->levelA->levelB->childComplex->childZ = "zz"

  putmess "Original Struct:"
  putmess vStruct->$dbgstring
  ; Result:
  ; Original Struct:
  ;  []
  ;    [levelA]
  ;      [levelB]
  ;        [childX] = "xx"
  ;        [childY] = "yy"
  ;          [childComplex]
  ;          [childZ] = "zz"

  ; To remove level B, so that all its children become children of levelA.
  ; 1. Assign 'vStruct1->levelA' as parent of levelB's children:
  ; 2. Reset the parent of 'vStruct1->levelA->levelB'
  vStruct->levelA->levelB->*->$parent = vStruct->levelA
  vStruct->levelA->levelB->$parent = ""

  putmess "Struct with LevelB removed:"
  putmess vStruct->$dbgstring
  ; Result:
  ;  Struct with LevelB removed:
  ;  []
  ;    [levelA]
  ;      [childX] = "xx"
  ;      [childY] = "yy"
```

```
  ;      [childComplex]
  ;      [childZ] = "zz"

  ; Note: LevelB is no longer part of a containing struct,
  ;       and no struct variables refer to levelB,
  ;       so the Struct is actually deleted: it is removed from memory.

end ;- entry REMOVE_STRUCT_LEVEL
; ========================================================================
;                            INSERT_STRUCT_LEVEL
; ========================================================================
; This entry shows how to insert an additional level into a Struct
; (the reverse of REMOVE_STRUCT_LEVEL))

entry INSERT_STRUCT_LEVEL
variables
  struct vStruct, vNewLevel
endvariables
; Display entry header in the message frame
  call printHeader("INSERT_STRUCT_LEVEL")

  ; Build a Struct
  vStruct->levelA = $newstruct
  vStruct->levelA->childX = "xx"
  vStruct->levelA->childY = "yy"
  vStruct->levelA->childComplex = $newstruct
  vStruct->levelA->childComplex->childZ = "zz"

  putmess "Original Struct"
  putmess vStruct->$dbgstring
  ; Result:
  ;  []
  ;    [levelA]
  ;      [childX] = "xx"
  ;      [childY] = "yy"
  ;      [childComplex]
  ;        [childZ] = "zz"


  ; Insert a new Struct level between 'vStruct1->levelA'
  ; and its children
  vNewLevel = $newstruct
  vStruct->levelA->*->$parent = vNewLevel
  vStruct->levelA->levelB = vNewLevel

  putmess "Struct with new LevelB"
  putmess vStruct->$dbgstring
  ; Result:
  ;  []
  ;    [levelA]
  ;      [levelB]
  ;        [childX] = "xx"
  ;        [childY] = "yy"
  ;        [childComplex]
  ;          [childZ] = "zz"
```

```
end ;- entry INSERT_STRUCT_LEVEL
; ========================================================================
;                            STRUCT_CONVERSIONS
; ========================================================================
entry STRUCT_CONVERSIONS
; This entry uses Struct manipulation to transform incoming XML
; data into Uniface entities so that it can be updated, (e.g. in code,
; by users, DB I/O, etc.), and transform it back to the original
; XML format.

; To run this example, you need to paint the following entities
; in your test component:
;  * Outer entity: BOOKS.MODELX,  with the fields: TITLE, CATEGORY, DESCRIPTION
;  * Inner entity: AUTHOR.MODELX, with the field : NAME

variables
  struct vStruct, vOutStruct
  string vOutXML
  numeric I
  string vWarning
endvariables
; Display entry header in the message frame
  call printHeader("STRUCT_CONVERSIONS")

  putmess "The starting point is the following XML sample:%%^%%%"
  ; $samplexml is defined as blockdata at the end of this entry
  putmess $samplexml

  xmlToStruct vStruct, $samplexml
  ; Result:
  ;          The starting point is the following XML sample:
  ;
  ;          <?xml version="1.0"?>
  ;          <catalog>
  ;             <book id="ref2451">
  ;                <author>Smith, John</author>
  ;                <title>My Road</title>
  ;                <genre>Biography</genre>
  ;                <publisher>ABC books</publisher>
  ;                <publication_date>2010-07-21</publication_date>
  ;                <description>Autobiography by John Smith</description>
  ;                <price>22.90</price>
  ;             </book>
  ;             <book id="ref7836">
  ;                <author>Black, E.J.</author>
  ;                <title>Summer recipes</title>
  ;                <genre>Cookery</genre>
  ;                <publisher>Black Bee Publishers</publisher>
  ;                <publication_date>2010-08-12</publication_date>
  ;                <description>Recipes by top chef E.J. Black</description>
  ;                <price>15.00</price>
  ;             </book>
  ;          </catalog>

  ; The XML tags can be relevant when converting back to XML,
  ; but are distracting when debugging the Struct.
```

```
  ; To display the Struct without tags, use $dbgstringplain.

  putmess "Original struct, as converted from XML (without tags):"
  putmess vStruct->$dbgstringplain
   ; Alternatively,
  ;    1. You could use the following code to remove tags:
  ;vStruct->$tags->*->$parent = "" ; collection update: all vStruct->$tags->*
  ;vStruct->*->$tags->*->$parent = ""       ; collecion in collection...
  ;vStruct->*->*->$tags->*->$parent = ""    ; etc.
  ;vStruct->*->*->*->$tags->*->$parent = ""
  ;    2. Or you could call the REMOVE_TAGS() entry
  ;call REMOVE_TAGS(vStruct)

  ; Result:
  ;          Original struct, as converted from XML (all $tags reset):
  ;          []
  ;          [catalog]
  ;            [book]
  ;              [id] = "ref2451"
  ;              [author] = "Smith, John"
  ;              [title] = "My Road"
  ;              [genre] = "Biography"
  ;              [publisher] = "ABC books"
  ;              [publication_date] = "2010-07-21"
  ;              [description] = "Autobiography by John Smith"
  ;              [price] = "22.90"
  ;            [book]
  ;              [id] = "ref7836"
  ;              [author] = "Black, E.J."
  ;              [title] = "Summer recipes"
  ;              [genre] = "Cookery"
  ;              [publisher] = "Black Bee Publishers"
  ;              [publication_date] = "2010-08-12"
  ;              [description] = "Recipes by top chef E.J. Black"
  ;              [price] = "15.00"

; Transform the Struct to match the required Uniface component structure

; Note: The XML includes elements that are not required in the data
;       structure of the component, such as publisher and price.
;       You can choose to remove the members that correspond to
;       these elements, but in this example, they are left in the Struct.
;       This will generate warnings in $procreturncontext

  ; 1. Change the structure for the outer entity
  ; FROM: vStruct->catalog->book  TO: vStruct->catalog->BOOKS->OCC

  ; - Add one more level BOOKS
  vStruct->catalog->BOOKS = $newstruct

  ; - Put all 'book' Structs in it
  vStruct->catalog->book->$parent = vStruct->catalog->BOOKS

  ; - Rename the 'book Structs to "OCC"
  vStruct->catalog->BOOKS->book->$name = "OCC"
```

```
; 2. Rename fields to match those in BOOKS, where necessary
vStruct->catalog->BOOKS->OCC->genre->$name = "CATEGORY"

; 3. Change the structure for the inner entity
; - Add a new level for the inner entity AUTHOR
vStruct->catalog->BOOKS->OCC->AUTHOR = $newstruct
vStruct->catalog->BOOKS->OCC->AUTHOR->OCC = $newstruct

; - For each BOOKS occurrence, move the author to this new Struct:
I = 1
while (I <= vStruct->catalog->BOOKS->OCC->$collsize)
  vStruct->catalog->BOOKS->OCC{I}->author->$parent =        %\
                vStruct->catalog->BOOKS->OCC{I}->AUTHOR->OCC
  I = I + 1
endwhile

; - Rename the XML fields to match those in the entity AUTHOR
vStruct->catalog->BOOKS->OCC->AUTHOR->OCC->author->$name = "NAME"

putmess "Struct modified to match component structure:"
putmess vStruct->$dbgstringplain
; Result:
;          Struct modified to match component structure:
;          []
;          [catalog]
;            [BOOKS]
;              [OCC]
;                [id] = "ref2451"
;                [title] = "My Road"
;                [CATEGORY] = "Biography"
;                [publisher] = "ABC books"
;                [publication_date] = "2010-07-21"
;                [description] = "Autobiography by John Smith"
;                [price] = "22.90"
;                [AUTHOR]
;                  [OCC]
;                    [NAME] = "Smith, John"
;              [OCC]
;                [id] = "ref7836"
;                [title] = "Summer recipes"
;                [CATEGORY] = "Cookery"
;                [publisher] = "Black Bee Publishers"
;                [publication_date] = "2010-08-12"
;                [description] = "Recipes by top chef E.J. Black"
;                [price] = "15.00"
;                [AUTHOR]
;                  [OCC]
;                    [NAME] = "Black, E.J."


; At this point, the component entities and fields become important,
; because the Struct data needs to be inserted into the component.
structToComponent vStruct->catalog->BOOKS
; Warnings are generated for entities and fields not found in the component:
if ($procerror < 0)
  putmess "Failed to execute 'structToComponent': ProcScript error:%%$procerror%%%"
```

```
  putmess $procerrorcontext
else
  ;Warnings are generated for unrecognized fields:
  if ($item("WARNINGS", $procreturncontext) > 0)
    putmess "Warnings after execution of 'structToComponent':"
    I = 1
    while (I <= $item("WARNINGS", $procreturncontext))
      vWarning = $itemnr(I, $item("DETAILS", $procreturncontext))
      putmess "  Warning %%i%%:"
      while (vWarning != "")
        putmess "    %%$itemnr(1, vWarning)%%%"
        delitem vWarning, 1
      endwhile
      I = I + 1
    endwhile
  endif
endif
; Result:
;       Warnings after execution of 'structToComponent':
;         Warning 1:
;           SEVERITY=Warning
;           ID=-1161
;           MNEM=<USTRUCTERR_NO_MATCHING_NAME>
;           DESCRIPTION=No matching name found during conversion from struct
;           CURRENTSTRUCT=BOOKS->OCC{1}->id{1}
;           ADDITIONAL=SPECIFIEDNAME=id•;EXPECTEDTYPE=entity or field
;         Warning 2:
;           ...etc...

; At this point regular Uniface processing may take place,
; This can be ProcScript processing, or a user that modifies data
; in the form. After that:

; Once any changes have been made, the data can be converted back
  componentToStruct vOutStruct

; Once again, to restrict the output in the message frame,
; you can use $dbgstringplain or remove the tags:
; call REMOVE_TAGS(vOutStruct)

putmess "%%^%%%"
putmess "Struct after conversion from component:"
putmess "(This requires that the required entities/fields are present in the component)"
putmess vOutStruct->$dbgstringplain
; Result:
;          [STRUCTSAMPLES]
;            [BOOKS.MODELX]
;              [OCC]
;                [TITLE] = "My Road"
;                [CATEGORY] = "Biography"
;                [DESCRIPTION] = "Autobiography by John Smith"
;                [AUTHOR.MODELX]
;                  [OCC]
;                    [NAME] = "Smith, John"
;              [OCC]
;                [TITLE] = "Summer recipes"
```

```
  ;                [CATEGORY] = "Cookery"
  ;                [DESCRIPTION] = "Recipes by top chef E.J. Black"
  ;                [AUTHOR.MODELX]
  ;                  [OCC]
  ;                    [NAME] = "Black, E.J."


; Reconstruct the original XML format:

  ; Move the author details from the AUTHOR level up to the BOOK level:
  ; 1. Rename the 'name' field (could also be done after the move)
  vOutStruct->BOOKS.MODELX->OCC->AUTHOR->OCC->name->$name = "author"

  ; 2. Move each occurrence up one level:
  I = 1
  while (I <= vOutStruct->BOOKS.MODELX->OCC->$collsize)
    vOutStruct->BOOKS.MODELX->OCC{I}->AUTHOR->OCC->name->$parent =       %\
                                          vOutStruct->BOOKS.MODELX->OCC{I}
    I = I + 1
  endwhile

  ; 3. Detach AUTHOR from its parent:
  vOutStruct->BOOKS.MODELX->OCC->AUTHOR.MODELX->$parent = "" ;- coll. upd.


  ; 4. Rename the field Structs:
  vOutStruct->BOOKS.MODELX->$name = "catalog" ;- use BOOKS for the "catalog"
  vOutStruct->catalog->OCC->$name = "book" ;- collection update
  vOutStruct->catalog->book->TITLE->$name = "title" ;- just the case
  vOutStruct->catalog->book->CATEGORY->$name = "genre" ;- collection update
  vOutStruct->catalog->book->CATEGORY->$name = "author" ;- collection update
  vOutStruct->catalog->book->DESCRIPTION->$name = "description" ;- the case

  ; 5 Add the XML declaration:
  vOutStruct->$tags->xmlVersion="1.0"

  ; 6. Make the top-level struct nameless, so that it
  ;    will not be included as an additional level in the XML:
  vOutStruct->$name = ""

  ; Convert the Struct to XML:
  structToXml vOutXML, vOutStruct
  putmess "The XML after conversion back from the component"
  putmess vOutXML
  ; result:
  ;          <?xml version="1.0"?>
  ;          <catalog>
  ;           <book>
  ;            <title>My Road</title>
  ;            <genre>Biography</genre>
  ;            <description>Autobiography by John Smith</description>
  ;           </book>
  ;           <book>
  ;            <title>Summer recipes</title>
  ;            <genre>Cookery</genre>
  ;            <description>Recipes by top chef E.J. Black</description>
```

```
  ;           </book>
  ;          </catalog>

 ; Note: Some xml elements, such as  <publisher>, are now lost,
 ;       but this was a choice.

 ; Assuming the component is a form, use the edit statement to see the data:
 ;edit


samplexml:blockdata @
<?xml version="1.0"?>
<catalog>
   <book id="ref2451">
      <author>Smith, John</author>
      <title>My Road</title>
      <genre>Biography</genre>
      <publisher>ABC books</publisher>
      <publication_date>2010-07-21</publication_date>
      <description>Autobiography by John Smith</description>
      <price>22.90</price>
   </book>
   <book id="ref7836">
      <author>Black, E.J.</author>
      <title>Summer recipes</title>
      <genre>Cookery</genre>
      <publisher>Black Bee Publishers</publisher>
      <publication_date>2010-08-12</publication_date>
      <description>Recipes by top chef E.J. Black</description>
      <price>15.00</price>
   </book>
</catalog>
@
end ;- function STRUCT_CONVERSIONS
```