Free Component Library (FCL): Reference guide.

Reference guide for FCL units.
Document version 2.1
November 2009

Michaël Van Canneyt

Contents

About this guide

This document describes all constants, types, variables, functions and procedures as they are declared in the units that come standard with the FCL (Free Component Library).

Throughout this document, we will refer to functions, types and variables with `typewriter` font. Functions and procedures gave their own subsections, and for each function or procedure we have the following topics:

Declaration The exact declaration of the function.

Description What does the procedure exactly do ?

Errors What errors can occur.

See Also Cross references to other related functions/commands.

0.1 Overview

The Free Component Library is a series of units that implemenent various classes and non-visual components for use with Free Pascal. They are building blocks for non-visual and visual programs, such as designed in Lazarus.

The `TDataset` descendents have been implemented in a way that makes them compatible to the Delphi implementation of these units. There are other units that have counterparts in Delphi, but most of them are unique to Free Pascal.

Chapter 1

Reference for unit 'ascii85'

1.1 Used units

Table 1.1: Used units by unit 'ascii85'

Name	Page
Classes	**??**
sysutils	**??**

1.2 Overview

The ascii85 provides an ASCII 85 or base 85 decoding algorithm. It is class and stream based: the TASCII85DecoderStream (38) stream can be used to decode any stream with ASCII85 encoded data.

Currently, no ASCII85 encoder stream is available.

It's usage and purpose is similar to the IDEA (225) or base64 (57) units.

1.3 Constants, types and variables

1.3.1 Types

```
TASCII85State = (ascInitial,ascOneEncodedChar,ascTwoEncodedChars,
                 ascThreeEncodedChars,ascFourEncodedChars,
                 ascNoEncodedChar,ascPrefix)
```

`TASCII85State` is for internal use, it contains the current state of the decoder.

1.4 TASCII85DecoderStream

1.4.1 Description

`TASCII85DecoderStream` is a read-only stream: it takes an input stream with ASCII 85 encoded data, and decodes the data as it is read. To this end, it overrides the TSTream.Read (**??**) method.

Table 1.2: Enumeration values for type TASCII85State

Value	Explanation
ascFourEncodedChars	Four encoded characters in buffer.
ascInitial	Initial state
ascNoEncodedChar	No encoded characters in buffer.
ascOneEncodedChar	One encoded character in buffer.
ascPrefix	Prefix processing
ascThreeEncodedChars	Three encoded characters in buffer.
ascTwoEncodedChars	Two encoded characters in buffer.

The stream cannot be written to, trying to write to the stream will result in an exception.

1.4.2 Method overview

Page	Property	Description
40	Close	Close decoder
40	ClosedP	Check if the state is correct
39	Create	Create new ASCII 85 decoder stream
39	Decode	Decode source byte
40	Destroy	Clean up instance
40	Read	Read data from stream
41	Seek	Set stream position

1.4.3 Property overview

Page	Property	Access	Description
41	BExpectBoundary	rw	Expect čharacter

1.4.4 TASCII85DecoderStream.Create

Synopsis: Create new ASCII 85 decoder stream

Declaration: `constructor Create(aStream: TStream)`

Visibility: published

Description: `Create` instantiates a new `TASCII85DecoderStream` instance, and sets `aStream` as the source stream.

See also: TASCII85DecoderStream.Destroy (40)

1.4.5 TASCII85DecoderStream.Decode

Synopsis: Decode source byte

Declaration: `procedure Decode(aInput: Byte)`

Visibility: published

Description: `Decode` decodes a source byte, and transfers it to the buffer. It is an internal routine and should not be used directly.

See also: TASCII85DecoderStream.Close (40)

1.4.6 TASCII85DecoderStream.Close

Synopsis: Close decoder

Declaration: `procedure Close`

Visibility: published

Description: `Close` closes the decoder mechanism: it checks if all data was read and performs a check to see whether all input data was consumed.

Errors: If the input stream was invalid, an `EConvertError` exception is raised.

See also: TASCII85DecoderStream.ClosedP (40), TASCII85DecoderStream.Read (40), TASCII85DecoderStream.Destroy (40)

1.4.7 TASCII85DecoderStream.ClosedP

Synopsis: Check if the state is correct

Declaration: `function ClosedP : Boolean`

Visibility: published

Description: `ClosedP` checks if the decoder state is one of `ascInitial`, `ascNoEncodedChar`, `ascPrefix`, and returns `True` if it is.

See also: TASCII85DecoderStream.Close (40), TASCII85DecoderStream.BExpectBoundary (41)

1.4.8 TASCII85DecoderStream.Destroy

Synopsis: Clean up instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` closes the input stream using Close (40) and cleans up the `TASCII85DecoderStream` instance from memory.

Errors: In case the input stream was invalid, an exception may occur.

See also: TASCII85DecoderStream.Close (40)

1.4.9 TASCII85DecoderStream.Read

Synopsis: Read data from stream

Declaration: `function Read(var aBuffer;aCount: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` attempts to read `aCount` bytes from the stream and places them in `aBuffer`. It reads only as much data as is available. The actual number of read bytes is returned.

The read method reads as much data from the input stream as needed to get to `aCount` bytes, in general this will be `aCount*5/4` bytes.

1.4.10 TASCII85DecoderStream.Seek

Synopsis: Set stream position

Declaration:
```
function Seek(aOffset: LongInt;aOrigin: Word) : LongInt;  Override
function Seek(const aOffset: Int64;aOrigin: TSeekOrigin) : Int64
             ;  Override;  Overload
```

Visibility: public

Description: `Seek` sets the stream position. It only allows to set the position to the current position of this file, and returns then the current position. All other arguments will result in an `EReadError` exception.

Errors: In case the arguments are different from `soCurrent` and 0, an `EReadError` exception will be raised.

See also: TASCII85DecoderStream.Read (40)

1.4.11 TASCII85DecoderStream.BExpectBoundary

Synopsis: Expect c̆haracter

Declaration:
```
Property BExpectBoundary : Boolean
```

Visibility: published

Access: Read,Write

Description: `BExpectBoundary` is `True` if a encoded data boundary is to be expected ("~>").

See also: TASCII85DecoderStream.ClosedP (40)

1.5 TASCII85EncoderStream

1.5.1 Method overview

Page	Property	Description
41	Create	
42	Destroy	
42	Write	

1.5.2 Property overview

Page	Property	Access	Description
42	Boundary	r	
42	Width	r	

1.5.3 TASCII85EncoderStream.Create

Declaration:
```
constructor Create(ADest: TStream;AWidth: Integer;ABoundary: Boolean)
```

Visibility: public

1.5.4 TASCII85EncoderStream.Destroy

Declaration: `destructor Destroy;  Override`

Visibility: public

1.5.5 TASCII85EncoderStream.Write

Declaration: `function Write(const aBuffer;aCount: LongInt) : LongInt;  Override`

Visibility: public

1.5.6 TASCII85EncoderStream.Width

Declaration: `Property Width : Integer`

Visibility: public

Access: Read

1.5.7 TASCII85EncoderStream.Boundary

Declaration: `Property Boundary : Boolean`

Visibility: public

Access: Read

1.6 TASCII85RingBuffer

1.6.1 Description

`TASCII85RingBuffer` is an internal buffer class: it maintains a memory buffer of 1Kb, for faster reading of the stream. It should not be necessary to instantiate an instance of this class, the TASCII85DecoderStream (38) decoder stream will create an instance of this class automatically.

1.6.2 Method overview

Page	Property	Description
43	Read	Read data from the internal buffer
42	Write	Write data to the internal buffer

1.6.3 Property overview

Page	Property	Access	Description
43	FillCount	r	Number of bytes in buffer
43	Size	r	Size of buffer

1.6.4 TASCII85RingBuffer.Write

Synopsis: Write data to the internal buffer

Declaration: `procedure Write(const aBuffer;aSize: Cardinal)`

Visibility: published

Description: `Write` writes `aSize` bytes from `aBuffer` to the internal memory buffer. Only as much bytes are written as will fit in the buffer.

See also: TASCII85RingBuffer.FillCount (43), TASCII85RingBuffer.Read (43), TASCII85RingBuffer.Size (43)

1.6.5 TASCII85RingBuffer.Read

Synopsis: Read data from the internal buffer

Declaration: `function Read(var aBuffer;aSize: Cardinal) : Cardinal`

Visibility: published

Description: `Read` will read `aSize` bytes from the internal buffer and writes them to `aBuffer`. If not enough bytes are available, only as much bytes as available will be written. The function returns the number of bytes transferred.

See also: TASCII85RingBuffer.FillCount (43), TASCII85RingBuffer.Write (42), TASCII85RingBuffer.Size (43)

1.6.6 TASCII85RingBuffer.FillCount

Synopsis: Number of bytes in buffer

Declaration: `Property FillCount : Cardinal`

Visibility: published

Access: Read

Description: `FillCount` is the available amount of bytes in the buffer.

See also: TASCII85RingBuffer.Write (42), TASCII85RingBuffer.Read (43), TASCII85RingBuffer.Size (43)

1.6.7 TASCII85RingBuffer.Size

Synopsis: Size of buffer

Declaration: `Property Size : Cardinal`

Visibility: published

Access: Read

Description: `Size` is the total size of the memory buffer. This is currently hardcoded to 1024Kb.

See also: TASCII85RingBuffer.FillCount (43)

Chapter 2

Reference for unit 'AVL_Tree'

2.1 Used units

Table 2.1: Used units by unit 'AVL_Tree'

Name	Page
Classes	??
sysutils	??

2.2 Overview

The `avl_tree` unit implements a general-purpose AVL (balanced) tree class: the TAVLTree (44) class and it's associated data node class TAVLTreeNode (53).

2.3 TAVLTree

2.3.1 Description

`TAVLTree` maintains a balanced AVL tree. The tree consists of TAVLTreeNode (53) nodes, each of which has a `Data` pointer associated with it. The `TAVLTree` component offers methods to balance and search the tree.

By default, the list is searched with a simple pointer comparison algorithm, but a custom search mechanism can be specified in the OnCompare (52) property.

2.3.2 Method overview

Page	Property	Description
49	Add	Add a new node to the tree
50	Clear	Clears the tree
51	ConsistencyCheck	Check the consistency of the tree
52	Create	Create a new instance of `TAVLTree`
49	Delete	Delete a node from the tree
52	Destroy	Destroy the `TAVLTree` instance
45	Find	Find a data item in the tree.
47	FindHighest	Find the highest (rightmost) node in the tree.
46	FindKey	Find a data item in the tree using alternate compare mechanism
47	FindLeftMost	Find the node most left to a specified data node
48	FindLeftMostKey	Find the node most left to a specified key node
48	FindLeftMostSameKey	Find the node most left to a specified node with the same data
46	FindLowest	Find the lowest (leftmost) node in the tree.
47	FindNearest	Find the node closest to the data in the tree
47	FindPointer	Search for a data pointer
46	FindPrecessor	
48	FindRightMost	Find the node most right to a specified node
48	FindRightMostKey	Find the node most right to a specified key node
49	FindRightMostSameKey	Find the node most right of a specified node with the same data
46	FindSuccessor	Find successor to node
51	FreeAndClear	Clears the tree and frees nodes
51	FreeAndDelete	Delete a node from the tree and destroy it
50	MoveDataLeftMost	Move data to the nearest left element
50	MoveDataRightMost	Move data to the nearest right element
49	Remove	Remove a data item from the list.
50	RemovePointer	Remove a pointer item from the list.
52	ReportAsString	Return the tree report as a string
52	SetNodeManager	
51	WriteReportToStream	Write the contents of the tree consistency check to the stream

2.3.3 Property overview

Page	Property	Access	Description
53	Count	r	Number of nodes in the tree.
52	OnCompare	rw	Compare function used when comparing nodes

2.3.4 TAVLTree.Find

Synopsis: Find a data item in the tree.

Declaration: `function Find(Data: Pointer) : TAVLTreeNode`

Visibility: public

Description: `Find` uses the default OnCompare (52) comparing function to find the `Data` pointer in the tree. It returns the `TAVLTreeNode` instance that results in a successful compare with the `Data` pointer, or `Nil` if none is found.

The default `OnCompare` function compares the actual pointers, which means that by default `Find` will give the same result as FindPointer (47).

See also: TAVLTree.OnCompare (52), TAVLTree.FindKey (46)

2.3.5 TAVLTree.FindKey

Synopsis: Find a data item in the tree using alternate compare mechanism

Declaration:
```
function FindKey(Key: Pointer;OnCompareKeyWithData: TListSortCompare)
                        : TAVLTreeNode
```

Visibility: public

Description: `FindKey` uses the specified `OnCompareKeyWithData` comparing function to find the `Key` pointer in the tree It returns the `TAVLTreeNode` instance that matches the `Data` pointer, or `Nil` if none is found.

See also: TAVLTree.OnCompare (52), TAVLTree.Find (45)

2.3.6 TAVLTree.FindSuccessor

Synopsis: Find successor to node

Declaration:
```
function FindSuccessor(ANode: TAVLTreeNode) : TAVLTreeNode
```

Visibility: public

Description: `FindSuccessor` returns the successor to `ANode`: this is the leftmost node in the right subtree, or the leftmost node above the node `ANode`. This can of course be `Nil`.

This method is used when a node must be inserted at the rightmost position.

See also: TAVLTree.FindPrecessor (46), TAVLTree.MoveDataRightMost (50)

2.3.7 TAVLTree.FindPrecessor

Synopsis:

Declaration:
```
function FindPrecessor(ANode: TAVLTreeNode) : TAVLTreeNode
```

Visibility: public

Description: `FindPrecessor` returns the successor to `ANode`: this is the rightmost node in the left subtree, or the rightmost node above the node `ANode`. This can of course be `Nil`.

This method is used when a node must be inserted at the leftmost position.

See also: TAVLTree.FindSuccessor (46), TAVLTree.MoveDataLeftMost (50)

2.3.8 TAVLTree.FindLowest

Synopsis: Find the lowest (leftmost) node in the tree.

Declaration:
```
function FindLowest : TAVLTreeNode
```

Visibility: public

Description: `FindLowest` returns the leftmost node in the tree, i.e. the node which is reached when descending from the rootnode via the left (**??**) subtrees.

See also: TAVLTree.FindHighest (47)

2.3.9 TAVLTree.FindHighest

Synopsis: Find the highest (rightmost) node in the tree.

Declaration: `function FindHighest : TAVLTreeNode`

Visibility: public

Description: `FindHighest` returns the rightmost node in the tree, i.e. the node which is reached when descending from the rootnode via the Right (**??**) subtrees.

See also: TAVLTree.FindLowest (46)

2.3.10 TAVLTree.FindNearest

Synopsis: Find the node closest to the data in the tree

Declaration: `function FindNearest(Data: Pointer) : TAVLTreeNode`

Visibility: public

Description: `FindNearest` searches the node in the data tree that is closest to the specified `Data`. If `Data` appears in the tree, then its node is returned.

See also: TAVLTree.FindHighest (47), TAVLTree.FindLowest (46), TAVLTree.Find (45), TAVLTree.FindKey (46)

2.3.11 TAVLTree.FindPointer

Synopsis: Search for a data pointer

Declaration: `function FindPointer(Data: Pointer) : TAVLTreeNode`

Visibility: public

Description: `FindPointer` searches for a node where the actual data pointer equals `Data`. This is a more fine search than find (45), where a custom compare function can be used.

The default OnCompare (52) compares the data pointers, so the default `Find` will return the same node as `FindPointer`

See also: TAVLTree.Find (45), TAVLTree.FindKey (46)

2.3.12 TAVLTree.FindLeftMost

Synopsis: Find the node most left to a specified data node

Declaration: `function FindLeftMost(Data: Pointer) : TAVLTreeNode`

Visibility: public

Description: `FindLeftMost` finds the node most left from the `Data` node. It starts at the preceding node for `Data` and tries to move as far right in the tree as possible.

This operation corresponds to finding the previous item in a list.

See also: TAVLTree.FindRightMost (48), TAVLTree.FindLeftMostKey (48), TAVLTree.FindRightMostKey (48)

2.3.13 TAVLTree.FindRightMost

Synopsis: Find the node most right to a specified node

Declaration:
```
function FindRightMost(Data: Pointer) : TAVLTreeNode
```

Visibility: public

Description: `FindRightMost` finds the node most right from the `Data` node. It starts at the succeding node for `Data` and tries to move as far left in the tree as possible.

This operation corresponds to finding the next item in a list.

See also: TAVLTree.FindLeftMost (47), TAVLTree.FindLeftMostKey (48), TAVLTree.FindRightMostKey (48)

2.3.14 TAVLTree.FindLeftMostKey

Synopsis: Find the node most left to a specified key node

Declaration:
```
function FindLeftMostKey(Key: Pointer;
                         OnCompareKeyWithData: TListSortCompare)
                          : TAVLTreeNode
```

Visibility: public

Description: `FindLeftMostKey` finds the node most left from the node associated with `Key`. It starts at the preceding node for `Key` and tries to move as far left in the tree as possible.

See also: TAVLTree.FindLeftMost (47), TAVLTree.FindRightMost (48), TAVLTree.FindRightMostKey (48)

2.3.15 TAVLTree.FindRightMostKey

Synopsis: Find the node most right to a specified key node

Declaration:
```
function FindRightMostKey(Key: Pointer;
                          OnCompareKeyWithData: TListSortCompare)
                           : TAVLTreeNode
```

Visibility: public

Description: `FindRightMostKey` finds the node most left from the node associated with `Key`. It starts at the succeding node for `Key` and tries to move as far right in the tree as possible.

See also: TAVLTree.FindLeftMost (47), TAVLTree.FindRightMost (48), TAVLTree.FindLeftMostKey (48)

2.3.16 TAVLTree.FindLeftMostSameKey

Synopsis: Find the node most left to a specified node with the same data

Declaration:
```
function FindLeftMostSameKey(ANode: TAVLTreeNode) : TAVLTreeNode
```

Visibility: public

Description: `FindLefMostSameKey` finds the node most left from and with the same data as the specified node `ANode`.

See also: TAVLTree.FindLeftMost (47), TAVLTree.FindLeftMostKey (48), TAVLTree.FindRightMostSameKey (49)

2.3.17 TAVLTree.FindRightMostSameKey

Synopsis: Find the node most right of a specified node with the same data

Declaration:
```
function FindRightMostSameKey(ANode: TAVLTreeNode) : TAVLTreeNode
```

Visibility: public

Description: `FindRighMostSameKey` finds the node most right from and with the same data as the specified node `ANode`.

See also: TAVLTree.FindRightMost (48), TAVLTree.FindRightMostKey (48), TAVLTree.FindLeftMostSameKey (48)

2.3.18 TAVLTree.Add

Synopsis: Add a new node to the tree

Declaration:
```
procedure Add(ANode: TAVLTreeNode)
function Add(Data: Pointer) : TAVLTreeNode
```

Visibility: public

Description: `Add` adds a new `Data` or `Node` to the tree. It inserts the node so that the tree is maximally balanced by rebalancing the tree after the insert. In case a `data` pointer is added to the tree, then the node that was created is returned.

See also: TAVLTree.Delete (49), TAVLTree.Remove (49)

2.3.19 TAVLTree.Delete

Synopsis: Delete a node from the tree

Declaration:
```
procedure Delete(ANode: TAVLTreeNode)
```

Visibility: public

Description: `Delete` removes the node from the tree. The node is not freed, but is passed to a TAVLTreeNode-MemManager (53) instance for future reuse. The data that the node represents is also not freed.

The tree is rebalanced after the node was deleted.

See also: TAVLTree.Remove (49), TAVLTree.RemovePointer (50), TAVLTree.Clear (50)

2.3.20 TAVLTree.Remove

Synopsis: Remove a data item from the list.

Declaration:
```
procedure Remove(Data: Pointer)
```

Visibility: public

Description: `Remove` finds the node associated with `Data` using find (45) and, if found, deletes it from the tree. Only the first occurrence of `Data` will be removed.

See also: TAVLTree.Delete (49), TAVLTree.RemovePointer (50), TAVLTree.Clear (50), TAVLTree.Find (45)

2.3.21 TAVLTree.RemovePointer

Synopsis: Remove a pointer item from the list.

Declaration: `procedure RemovePointer(Data: Pointer)`

Visibility: public

Description: `Remove` uses FindPointer (47) to find the node associated with the pointer `Data` and, if found, deletes it from the tree. Only the first occurrence of `Data` will be removed.

See also: TAVLTree.Remove (49), TAVLTree.Delete (49), TAVLTree.Clear (50)

2.3.22 TAVLTree.MoveDataLeftMost

Synopsis: Move data to the nearest left element

Declaration: `procedure MoveDataLeftMost(var ANode: TAVLTreeNode)`

Visibility: public

Description: `MoveDataLeftMost` moves the data from the node `ANode` to the nearest left location relative to `Anode`. It returns the new node where the data is positioned. The data from the former left node will be switched to `ANode`.

This operation corresponds to switching the current with the previous element in a list.

See also: TAVLTree.MoveDataRightMost (50)

2.3.23 TAVLTree.MoveDataRightMost

Synopsis: Move data to the nearest right element

Declaration: `procedure MoveDataRightMost(var ANode: TAVLTreeNode)`

Visibility: public

Description: `MoveDataRightMost` moves the data from the node `ANode` to the rightmost location relative to `Anode`. It returns the new node where the data is positioned. The data from the former rightmost node will be switched to `ANode`.

This operation corresponds to switching the current with the next element in a list.

See also: TAVLTree.MoveDataLeftMost (50)

2.3.24 TAVLTree.Clear

Synopsis: Clears the tree

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` deletes all nodes from the tree. The nodes themselves are not freed, and the data pointer in the nodes is also not freed.

If the node's data must be freed as well, use TAVLTree.FreeAndClear (51) instead.

See also: TAVLTree.FreeAndClear (51), TAVLTree.Delete (49)

2.3.25 TAVLTree.FreeAndClear

Synopsis: Clears the tree and frees nodes

Declaration: `procedure FreeAndClear`

Visibility: public

Description: `FreeAndClear` deletes all nodes from the tree. The data pointer in the nodes is assumed to be an object, and is freed prior to deleting the node from the tree.

See also: TAVLTree.Clear (50), TAVLTree.Delete (49), TAVLTree.FreeAndDelete (51)

2.3.26 TAVLTree.FreeAndDelete

Synopsis: Delete a node from the tree and destroy it

Declaration: `procedure FreeAndDelete(ANode: TAVLTreeNode)`

Visibility: public

Description: `FreeAndDelete` deletes a node from the tree, and destroys the data pointer: The data pointer in the nodes is assumed to be an object, and is freed by calling its destructor.

See also: TAVLTree.Clear (50), TAVLTree.Delete (49), TAVLTree.FreeAndClear (51)

2.3.27 TAVLTree.ConsistencyCheck

Synopsis: Check the consistency of the tree

Declaration: `function ConsistencyCheck : Integer`

Visibility: public

Description: `ConsistencyCheck` checks the correctness of the tree. It returns 0 if the tree is internally consistent, and a negative number if the tree contais an error somewhere.

-1The Count property doesn't match the actual node count

-2A left node does not point to the correct parent

-3A left node is larger than parent node

-4A right node does not point to the correct parent

-5A right node is less than parent node

-6The balance of a node is not calculated correctly

See also: TAVLTree.WriteReportToStream (51)

2.3.28 TAVLTree.WriteReportToStream

Synopsis: Write the contents of the tree consistency check to the stream

Declaration: `procedure WriteReportToStream(s: TStream;var StreamSize: Int64)`

Visibility: public

Description: `WriteReportToStream` writes a visual representation of the tree to the stream `S`. The total number of written bytes is returnes in `StreamSize`. This method is only useful for debugging purposes.

See also: TAVLTree.ConsistencyCheck (51)

2.3.29 TAVLTree.ReportAsString

Synopsis: Return the tree report as a string

Declaration: `function ReportAsString : String`

Visibility: public

Description: `ReportAsString` calls WriteReportToStream (51) and retuns the stream data as a string.

See also: TAVLTree.WriteReportToStream (51)

2.3.30 TAVLTree.SetNodeManager

Declaration: `procedure SetNodeManager(newmgr: TBaseAVLTreeNodeManager)`

Visibility: public

2.3.31 TAVLTree.Create

Synopsis: Create a new instance of `TAVLTree`

Declaration: `constructor Create(OnCompareMethod: TListSortCompare)`
`constructor Create`

Visibility: public

Description: `Create` initializes a new instance of TAVLTree (44). An alternate OnCompare (44) can be provided: the default `OnCompare` method compares the 2 data pointers of a node.

See also: OnCompare (44)

2.3.32 TAVLTree.Destroy

Synopsis: Destroy the `TAVLTree` instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` clears the nodes (the node data is not freed) and then destroys the `TAVLTree` instance.

See also: TAVLTree.Create (52), TAVLTree.Clean (44)

2.3.33 TAVLTree.OnCompare

Synopsis: Compare function used when comparing nodes

Declaration: `Property OnCompare : TListSortCompare`

Visibility: public

Access: Read,Write

Description: `OnCompare` is the comparing function used when the data of 2 nodes must be compared. By default, the function simply compares the 2 data pointers. A different function can be specified on creation.

See also: TAVLTree.Create (52)

2.3.34 TAVLTree.Count

Synopsis: Number of nodes in the tree.

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the number of nodes in the tree.

2.4 TAVLTreeNode

2.4.1 Description

`TAVLTreeNode` represents a single node in the AVL tree. It contains references to the other nodes in the tree, and provides a Data (**??**) pointer which can be used to store the data, associated with the node.

2.4.2 Method overview

Page	Property	Description
53	Clear	Clears the node's data
53	TreeDepth	Level of the node in the tree below

2.4.3 TAVLTreeNode.Clear

Synopsis: Clears the node's data

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` clears all pointers and references in the node. It does not free the memory pointed to by these references.

2.4.4 TAVLTreeNode.TreeDepth

Synopsis: Level of the node in the tree below

Declaration: `function TreeDepth : Integer`

Visibility: public

Description: `TreeDepth` is the height of the node: this is the largest height of the left or right nodes, plus 1. If no nodes appear below this node (`left` and `Right` are `Nil`), the depth is 1.

See also: TAVLTreeNode.Balance (**??**)

2.5 TAVLTreeNodeMemManager

2.5.1 Description

`TAVLTreeNodeMemManager` is an internal object used by the avl_tree unit. Normally, no instance of this object should be created: An instance is created by the unit initialization code, and freed when the unit is finalized.

2.5.2 Method overview

Page	Property	Description
54	Clear	Frees all unused nodes
55	Create	Create a new instance of `TAVLTreeNodeMemManager`
55	Destroy	
54	DisposeNode	Return a node to the free list
54	NewNode	Create a new `TAVLTreeNode` instance

2.5.3 Property overview

Page	Property	Access	Description
56	Count	r	Number of nodes in the list.
55	MaximumFreeNodeRatio	rw	Maximum amount of free nodes in the list
55	MinimumFreeNode	rw	Minimum amount of free nodes to be kept.

2.5.4 TAVLTreeNodeMemManager.DisposeNode

Synopsis: Return a node to the free list

Declaration: `procedure DisposeNode(ANode: TAVLTreeNode); Override`

Visibility: public

Description: `DisposeNode` is used to put the node `ANode` in the list of free nodes, or optionally destroy it if the free list is full. After a call to `DisposeNode`, `ANode` must be considered invalid.

See also: TAVLTreeNodeMemManager.NewNode (54)

2.5.5 TAVLTreeNodeMemManager.NewNode

Synopsis: Create a new `TAVLTreeNode` instance

Declaration: `function NewNode : TAVLTreeNode; Override`

Visibility: public

Description: `NewNode` returns a new TAVLTreeNode (53) instance. If there is a node in the free list, itare returned. If no more free nodes are present, a new node is created.

See also: TAVLTreeNodeMemManager.DisposeNode (54)

2.5.6 TAVLTreeNodeMemManager.Clear

Synopsis: Frees all unused nodes

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` removes all unused nodes from the list and frees them.

See also: TAVLTreeNodeMemManager.MinimumFreeNode (55), TAVLTreeNodeMemManager.MaximumFreeNodeRatio (55)

2.5.7 TAVLTreeNodeMemManager.Create

Synopsis: Create a new instance of `TAVLTreeNodeMemManager`

Declaration: `constructor Create`

Visibility: public

Description: `Create` initializes a new instance of `TAVLTreeNodeMemManager`.

See also: TAVLTreeNodeMemManager.Destroy (55)

2.5.8 TAVLTreeNodeMemManager.Destroy

Synopsis:

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` calls clear to clean up the free node list and then calls the inherited destroy.

See also: TAVLTreeNodeMemManager.Create (55)

2.5.9 TAVLTreeNodeMemManager.MinimumFreeNode

Synopsis: Minimum amount of free nodes to be kept.

Declaration: `Property MinimumFreeNode : Integer`

Visibility: public

Access: Read,Write

Description: `MinimumFreeNode` is the minimum amount of nodes that must be kept in the free nodes list.

See also: TAVLTreeNodeMemManager.MaximumFreeNodeRatio (55)

2.5.10 TAVLTreeNodeMemManager.MaximumFreeNodeRatio

Synopsis: Maximum amount of free nodes in the list

Declaration: `Property MaximumFreeNodeRatio : Integer`

Visibility: public

Access: Read,Write

Description: `MaximumFreeNodeRatio` is the maximum amount of free nodes that should be kept in the list: if a node is disposed of, then the ratio of the free nodes versus the total amount of nodes is checked, and if it is less than the `MaximumFreeNodeRatio` ratio but larger than the minimum amount of free nodes, then the node is disposed of instead of added to the free list.

See also: TAVLTreeNodeMemManager.Count (56), TAVLTreeNodeMemManager.MinimumFreeNode (55)

2.5.11 TAVLTreeNodeMemManager.Count

Synopsis: Number of nodes in the list.

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the total number of nodes in the list, used or not.

See also: TAVLTreeNodeMemManager.MinimumFreeNode (55), TAVLTreeNodeMemManager.MaximumFreeNodeRatio (55)

2.6 TBaseAVLTreeNodeManager

2.6.1 Method overview

Page	Property	Description
56	DisposeNode	
56	NewNode	

2.6.2 TBaseAVLTreeNodeManager.DisposeNode

Declaration: `procedure DisposeNode(ANode: TAVLTreeNode); Virtual; Abstract`

Visibility: public

2.6.3 TBaseAVLTreeNodeManager.NewNode

Declaration: `function NewNode : TAVLTreeNode; Virtual; Abstract`

Visibility: public

Chapter 3

Reference for unit 'base64'

3.1 Used units

Table 3.1: Used units by unit 'base64'

Name	Page
Classes	??
sysutils	??

3.2 Overview

base64 implements base64 encoding (as used for instance in MIME encoding) based on streams. it implements 2 streams which encode or decode anything written or read from it. The source or the destination of the encoded data is another stream. 2 classes are implemented for this: TBase64EncodingStream (60) for encoding, and TBase64DecodingStream (58) for decoding.

The streams are designed as plug-in streams, which can be placed between other streams, to provide base64 encoding and decoding on-the-fly...

3.3 Constants, types and variables

3.3.1 Types

```
TBase64DecodingMode = (bdmStrict,bdmMIME)
```

Table 3.2: Enumeration values for type TBase64DecodingMode

Value	Explanation
bdmMIME	MIME encoding
bdmStrict	Strict encoding

`TBase64DecodingMode` determines the decoding algorithm used by TBase64DecodingStream (58). There are 2 modes:

bdmStrict Strict mode, which follows RFC3548 and rejects any characters outside of base64 alphabet. In this mode only up to two '=' characters are accepted at the end. It requires the input to have a Size being a multiple of 4, otherwise an EBase64DecodingException (58) exception is raised.

bdmMime MIME mode, which follows RFC2045 and ignores any characters outside of base64 alphabet. In this mode any '=' is seen as the end of string, it handles apparently truncated input streams gracefully.

3.4 EBase64DecodingException

3.4.1 Description

`EBase64DecodeException` is raised when the stream contains errors against the encoding format. Whether or not this exception is raised depends on the mode in which the stream is decoded.

3.5 TBase64DecodingStream

3.5.1 Description

`TBase64DecodingStream` can be used to read data from a stream (the source stream) that contains Base64 encoded data. The data is read and decoded on-the-fly.

The decoding stream is read-only, and provides a limited forward-seek capability.

3.5.2 Method overview

Page	Property	Description
58	Create	Create a new instance of the `TBase64DecodingStream` class
59	Read	Read and decrypt data from the source stream
59	Reset	Reset the stream
59	Seek	Set stream position.

3.5.3 Property overview

Page	Property	Access	Description
60	EOF	r	
60	Mode	rw	Decoding mode

3.5.4 TBase64DecodingStream.Create

Synopsis: Create a new instance of the `TBase64DecodingStream` class

Declaration:
```
constructor Create(ASource: TStream)
constructor Create(ASource: TStream;AMode: TBase64DecodingMode)
```

Visibility: public

Description: `Create` creates a new instance of the `TBase64DecodingStream` class. It stores the source stream `ASource` for reading the data from.

The optional `AMode` parameter determines the mode in which the decoding will be done. If omitted, `bdmMIME` is used.

See also: TBase64EncodingStream.Create (60), TBase64DecodingMode (57)

3.5.5 TBase64DecodingStream.Reset

Synopsis: Reset the stream

Declaration: `procedure Reset`

Visibility: public

Description: `Reset` resets the data as if it was again on the start of the decoding stream.

Errors: None.

See also: TBase64DecodingStream.EOF (60), TBase64DecodingStream.Read (59)

3.5.6 TBase64DecodingStream.Read

Synopsis: Read and decrypt data from the source stream

Declaration: `function Read(var Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` reads encrypted data from the source stream and stores this data in `Buffer`. At most `Count` bytes will be stored in the buffer, but more bytes will be read from the source stream: the encoding algorithm multiplies the number of bytes.

The function returns the number of bytes stored in the buffer.

Errors: If an error occurs during the read from the source stream, an exception may occur.

See also: TBase64DecodingStream.Write (58), TBase64DecodingStream.Seek (59), #rtl.classes.TStream.Read (**??**)

3.5.7 TBase64DecodingStream.Seek

Synopsis: Set stream position.

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` sets the position of the stream. In the `TBase64DecodingStream` class, the `seek` operation is forward only, it does not support backward seeks. The forward seek is emulated by reading and discarding data till the desired position is reached.

For an explanation of the parameters, see TStream.Seek (**??**)

Errors: In case of an unsupported operation, an `EStreamError` exception is raised.

See also: TBase64DecodingStream.Read (59), TBase64DecodingStream.Write (58), TBase64EncodingStream.Seek (61), #rtl.classes.TStream.Seek (**??**)

3.5.8 TBase64DecodingStream.EOF

Synopsis:

Declaration: `Property EOF : Boolean`

Visibility: public

Access: Read

Description:

3.5.9 TBase64DecodingStream.Mode

Synopsis: Decoding mode

Declaration: `Property Mode : TBase64DecodingMode`

Visibility: public

Access: Read,Write

Description: `Mode` is the mode in which the stream is read. It can be set when creating the stream or at any time afterwards.

See also: TBase64DecodingStream (58)

3.6 TBase64EncodingStream

3.6.1 Description

`TBase64EncodingStream` can be used to encode data using the base64 algorithm. At creation time, a destination stream is specified. Any data written to the `TBase64EncodingStream` instance will be base64 encoded, and subsequently written to the destination stream.

The `TBase64EncodingStream` stream is a write-only stream. Obviously it is also not seekable. It is meant to be included in a chain of streams.

3.6.2 Method overview

Page	Property	Description
60	Create	Create a new instance of the `TBase64EncodingStream` class.
61	Destroy	Remove a `TBase64EncodingStream` instannce from memory
61	Seek	Position the stream
61	Write	Write data to the stream.

3.6.3 TBase64EncodingStream.Create

Synopsis: Create a new instance of the `TBase64EncodingStream` class.

Declaration: `constructor Create(ASource: TStream)`

Visibility: public

Description: `Create` instantiates a new `TBase64EncodingStream` class. The `ASource` stream is stored and used to write the encoded data to.

See also: TBase64EncodingStream.Destroy (61), TBase64DecodingStream.Create (58)

3.6.4 TBase64EncodingStream.Destroy

Synopsis: Remove a `TBase64EncodingStream` instannce from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` flushes any remaining output and then removes the `TBase64EncodingStream` instance from memory by calling the inherited destructor.

Errors: An exception may be raised if the destination stream no longer exists or is closed.

See also: TBase64EncodingStream.Create (60)

3.6.5 TBase64EncodingStream.Write

Synopsis: Write data to the stream.

Declaration: `function Write(const Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Write` encodes `Count` bytes from `Buffer` using the Base64 mechanism, and then writes the encoded data to the destination stream. It returns the number of bytes from `Buffer` that were actually written. Note that this is not the number of bytes written to the destination stream: the base64 mechanism writes more bytes to the destination stream.

Errors: If there is an error writing to the destination stream, an error may occur.

See also: TBase64EncodingStream.Seek (61), TBase64EncodingStream.Read (60), TBase64DecodingStream.Write (58), #rtl.classes.TStream.Write (**??**)

3.6.6 TBase64EncodingStream.Seek

Synopsis: Position the stream

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` always raises an `EStreamError` exception unless the arguments it received it don't change the current file pointer position. The encryption stream is not seekable.

Errors: An `EStreamError` error is raised.

See also: TBase64EncodingStream.Read (60), TBase64EncodingStream.Write (61), #rtl.classes.TStream.Seek (**??**)

Chapter 4

Reference for unit 'BlowFish'

4.1 Used units

Table 4.1: Used units by unit 'BlowFish'

Name	Page
Classes	??
sysutils	??

4.2 Overview

The BlowFish implements a class TBlowFish (63) to handle blowfish encryption/descryption of memory buffers, and 2 TStream (**??**) descendents TBlowFishDeCryptStream (64) which descrypts any data that is read from it on the fly, as well as TBlowFishEnCryptStream (65) which encrypts the data that is written to it on the fly.

4.3 Constants, types and variables

4.3.1 Constants

```
BFRounds = 16
```

Number of rounds in blowfish encryption.

4.3.2 Types

```
PBlowFishKey = ^TBlowFishKey
```

`PBlowFishKey` is a simple pointer to a TBlowFishKey (63) array.

```
TBFBlock = Array[0..1] of LongInt
```

TBFBlock is the basic data structure used by the encrypting/decrypting routines in TBlowFish (63), TBlowFishDeCryptStream (64) and TBlowFishEnCryptStream (65). It is the basic encryption/decryption block for all encrypting/decrypting: all encrypting/decrypting happens on a TBFBlock structure.

```
TBlowFishKey = Array[0..55] of Byte
```

TBlowFishKey is a data structure which keeps the encryption or decryption key for the TBlowFish (63), TBlowFishDeCryptStream (64) and TBlowFishEnCryptStream (65) classes. It should be filled with the encryption key and passed to the constructor of one of these classes.

4.4 EBlowFishError

4.4.1 Description

EBlowFishError is used by the TBlowFishStream (66), TBlowFishEncryptStream (65) and TBlowFishDecryptStream (64) classes to report errors.

4.5 TBlowFish

4.5.1 Description

TBlowFish is a simple class that can be used to encrypt/decrypt a single TBFBlock (63) data block with the Encrypt (63) and Decrypt (64) calls. It is used internally by the TBlowFishEnCryptStream (65) and TBlowFishDeCryptStream (64) classes to encrypt or decrypt the actual data.

4.5.2 Method overview

Page	Property	Description
63	Create	Create a new instance of the TBlowFish class
64	Decrypt	Decrypt a block
63	Encrypt	Encrypt a block

4.5.3 TBlowFish.Create

Synopsis: Create a new instance of the TBlowFish class

Declaration:
```
constructor Create(Key: TBlowFishKey;KeySize: Integer)
```

Visibility: public

Description: Create initializes a new instance of the TBlowFish class: it stores the key Key in the internal data structures so it can be used in later calls to Encrypt (63) and Decrypt (64).

See also: TBlowFish.Encrypt (63), TBlowFish.Decrypt (64)

4.5.4 TBlowFish.Encrypt

Synopsis: Encrypt a block

Declaration:
```
procedure Encrypt(var Block: TBFBlock)
```

Visibility: public

Description: `Encrypt` encrypts the data in `Block` (always 8 bytes) using the key (63) specified when the `TBlowFish` instance was created.

See also: TBlowFishKey (63), TBlowFish.Decrypt (64), TBlowFish.Create (63)

4.5.5 TBlowFish.Decrypt

Synopsis: Decrypt a block

Declaration: `procedure Decrypt(var Block: TBFBlock)`

Visibility: public

Description: `ncrypt` decrypts the data in `Block` (always 8 bytes) using the key (63) specified when the `TBlowFish` instance was created. The data must have been encrypted with the same key and the Encrypt (63) call.

See also: TBlowFishKey (63), TBlowFish.Encrypt (63), TBlowFish.Create (63)

4.6 TBlowFishDeCryptStream

4.6.1 Description

The `TBlowFishDecryptStream` provides On-the-fly Blowfish decryption: all data that is read from the source stream is decrypted before it is placed in the output buffer. The source stream must be specified when the `TBlowFishDecryptStream` instance is created. The Decryption key must also be created when the stream instance is created, and must be the same key as the one used when encrypting the data.

This is a read-only stream: it is seekable only in a forward direction, and data can only be read from it, writing is not possible. For writing data so it is encrypted, the TBlowFishEncryptStream (65) stream must be used.

4.6.2 Method overview

Page	Property	Description
64	Read	Read data from the stream
65	Seek	Set the stream position.

4.6.3 TBlowFishDeCryptStream.Read

Synopsis: Read data from the stream

Declaration: `function Read(var Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` reads `Count` bytes from the source stream, decrypts them using the key provided when the `TBlowFishDeCryptStream` instance was created, and writes the decrypted data to `Buffer`

See also: TBlowFishStream.Create (67), TBlowFishEncryptStream (65)

4.6.4 TBlowFishDeCryptStream.Seek

Synopsis: Set the stream position.

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` emulates a forward seek by reading and discarding data. The discarded data is lost. Since it is a forward seek, this means that only `soFromCurrent` can be specified for `Origin` with a positive (or zero) `Offset` value. All other values will result in an exception. The function returns the new position in the stream.

Errors: If any other combination of `Offset` and `Origin` than the allowed combination is specified, then an EBlowFishError (63) exception will be raised.

See also: TBlowFishDeCryptStream.Read (64), EBlowFishError (63)

4.7 TBlowFishEncryptStream

4.7.1 Description

The `TBlowFishEncryptStream` provides On-the-fly Blowfish encryption: all data that is written to it is encrypted and then written to a destination stream, which must be specified when the `TBlowFishEncryptStream` instance is created. The encryption key must also be created when the stream instance is created.

This is a write-only stream: it is not seekable, and data can only be written to it, reading is not possible. For reading encrypted data, the TBlowFishDecryptStream (64) stream must be used.

4.7.2 Method overview

Page	Property	Description
65	Destroy	Free the `TBlowFishEncryptStream`
66	Flush	Flush the encryption buffer
66	Seek	Set the position in the stream
65	Write	Write data to the stream

4.7.3 TBlowFishEncryptStream.Destroy

Synopsis: Free the `TBlowFishEncryptStream`

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` flushes the encryption buffer, and writes it to the destination stream. After that the `Inherited` destructor is called to clean up the `TBlowFishEncryptStream` instance.

See also: TBlowFishEncryptStream.Flush (66), TBlowFishStream.Create (67)

4.7.4 TBlowFishEncryptStream.Write

Synopsis: Write data to the stream

Declaration: `function Write(const Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Write` will encrypt and write `Count` bytes from `Buffer` to the destination stream. The function returns the actual number of bytes written. The data is not encrypted in-place, but placed in a special buffer for encryption.

Data is always written 4 bytes at a time, since this is the amount of bytes required by the Blowfish algorithm. If no multiple of 4 was written to the destination stream, the Flush (66) mechanism can be used to write the remaining bytes.

See also: TBlowFishEncryptStream.Read (65)

4.7.5 TBlowFishEncryptStream.Seek

Synopsis: Set the position in the stream

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Read` will raise an `EBlowFishError` exception: `TBlowFishEncryptStream` is a write-only stream, and cannot be positioned.

Errors: Calling this function always results in an EBlowFishError (63) exception.

See also: TBlowFishEncryptStream.Write (65)

4.7.6 TBlowFishEncryptStream.Flush

Synopsis: Flush the encryption buffer

Declaration: `procedure Flush`

Visibility: public

Description: `Flush` writes the remaining data in the encryption buffer to the destination stream.

For efficiency, data is always written 4 bytes at a time, since this is the amount of bytes required by the Blowfish algorithm. If no multiple of 4 was written to the destination stream, the `Flush` mechanism can be used to write the remaining bytes.

`Flush` is called automatically when the stream is destroyed, so there is no need to call it after all data was written and the stream is no longer needed.

See also: TBlowFishEncryptStream.Write (65), TBFBlock (63)

4.8 TBlowFishStream

4.8.1 Description

`TBlowFishStream` is an abstract class which is used as a parent class for TBlowFishEncrypt-Stream (65) and TBlowFishDecryptStream (64). It simply provides a constructor and storage for a TBlowFish (63) instance and for the source or destination stream.

Do not create an instance of `TBlowFishStream` directly. Instead create one of the descendent classes `TBlowFishEncryptStream` or `TBlowFishDecryptStream`.

4.8.2 Method overview

Page	Property	Description
67	Create	Create a new instance of the `TBlowFishStream` class
67	Destroy	Destroy the `TBlowFishStream` instance.

4.8.3 Property overview

Page	Property	Access	Description
67	BlowFish	r	Blowfish instance used when encrypting/descrypting

4.8.4 TBlowFishStream.Create

Synopsis: Create a new instance of the `TBlowFishStream` class

Declaration:
```
constructor Create(AKey: TBlowFishKey;AKeySize: Byte;Dest: TStream)
constructor Create(const KeyPhrase: String;Dest: TStream)
```

Visibility: public

Description: `Create` initializes a new instance of `TBlowFishStream`, and creates an internal instance of TBlowFish (63) using `AKey` and `AKeySize`. The `Dest` stream is stored so the descendent classes can refer to it.

Do not create an instance of `TBlowFishStream` directly. Instead create one of the descendent classes `TBlowFishEncryptStream` or `TBlowFishDecryptStream`.

The overloaded version with the `KeyPhrase` string argument is used for easy access: it computes the blowfish key from the given string.

See also: TBlowFishEncryptStream (65), TBlowFishDecryptStream (64), TBlowFish (63)

4.8.5 TBlowFishStream.Destroy

Synopsis: Destroy the `TBlowFishStream` instance.

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` cleans up the internal TBlowFish (63) instance.

Errors:

See also: TBlowFishStream.Create (67), TBlowFish (63)

4.8.6 TBlowFishStream.BlowFish

Synopsis: Blowfish instance used when encrypting/descrypting

Declaration:
```
Property BlowFish : TBlowFish
```

Visibility: public

Access: Read

Description: `BlowFish` is the TBlowFish (63) instance which is created when the `TBlowFishStream` class is initialized. Normally it should not be used directly, it's intended for access by the descencent classes TBlowFishEncryptStream (65) and TBlowFishDecryptStream (64).

See also: TBlowFishEncryptStream (65), TBlowFishDecryptStream (64), TBlowFish (63)

Chapter 5

Reference for unit 'bufstream'

5.1 Used units

Table 5.1: Used units by unit 'bufstream'

Name	Page
Classes	??
sysutils	??

5.2 Overview

BufStream implements two one-way buffered streams: the streams store all data from (or for) the source stream in a memory buffer, and only flush the buffer when it's full (or refill it when it's empty). The buffer size can be specified at creation time. 2 streams are implemented: TReadBufStream (72) which is for reading only, and TWriteBufStream (72) which is for writing only.

Buffered streams can help in speeding up read or write operations, especially when a lot of small read/write operations are done: it avoids doing a lot of operating system calls.

5.3 Constants, types and variables

5.3.1 Constants

```
DefaultBufferCapacity : Integer = 16
```

If no buffer size is specified when the stream is created, then this size is used.

5.4 TBufStream

5.4.1 Description

`TBufStream` is the common ancestor for the TReadBufStream (72) and TWriteBufStream (72) streams. It completely handles the buffer memory management and position management. An in-

stance of TBufStream should never be created directly. It also keeps the instance of the source stream.

5.4.2 Method overview

Page	Property	Description
70	Create	Create a new TBufStream instance.
70	Destroy	Destroys the TBufStream instance

5.4.3 Property overview

Page	Property	Access	Description
70	Buffer	r	The current buffer
71	BufferPos	r	Current buffer position.
71	BufferSize	r	Amount of data in the buffer
71	Capacity	rw	Current buffer capacity

5.4.4 TBufStream.Create

Synopsis: Create a new TBufStream instance.

Declaration:
```
constructor Create(ASource: TStream;ACapacity: Integer)
constructor Create(ASource: TStream)
```

Visibility: public

Description: Create creates a new TBufStream instance. A buffer of size ACapacity is allocated, and the ASource source (or destination) stream is stored. If no capacity is specified, then DefaultBufferCapacity (69) is used as the capacity.

An instance of TBufStream should never be instantiated directly. Instead, an instance of TReadBufStream (72) or TWriteBufStream (72) should be created.

Errors: If not enough memory is available for the buffer, then an exception may be raised.

See also: TBufStream.Destroy (70), TReadBufStream (72), TWriteBufStream (72)

5.4.5 TBufStream.Destroy

Synopsis: Destroys the TBufStream instance

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: Destroy destroys the instance of TBufStream. It flushes the buffer, deallocates it, and then destroys the TBufStream instance.

See also: TBufStream.Create (70), TReadBufStream (72), TWriteBufStream (72)

5.4.6 TBufStream.Buffer

Synopsis: The current buffer

Declaration:
```
Property Buffer : Pointer
```

Visibility: public

Access: Read

Description: `Buffer` is a pointer to the actual buffer in use.

See also: TBufStream.Create (70), TBufStream.Capacity (71), TBufStream.BufferSize (71)

5.4.7 TBufStream.Capacity

Synopsis: Current buffer capacity

Declaration: `Property Capacity : Integer`

Visibility: public

Access: Read,Write

Description: `Capacity` is the amount of memory the buffer occupies. To change the buffer size, the capacity can be set. Note that the capacity cannot be set to a value that is less than the current buffer size, i.e. the current amount of data in the buffer.

See also: TBufStream.Create (70), TBufStream.Buffer (70), TBufStream.BufferSize (71), TBufStream.BufferPos (71)

5.4.8 TBufStream.BufferPos

Synopsis: Current buffer position.

Declaration: `Property BufferPos : Integer`

Visibility: public

Access: Read

Description: `BufPos` is the current stream position in the buffer. Depending on whether the stream is used for reading or writing, data will be read from this position, or will be written at this position in the buffer.

See also: TBufStream.Create (70), TBufStream.Buffer (70), TBufStream.BufferSize (71), TBufStream.Capacity (71)

5.4.9 TBufStream.BufferSize

Synopsis: Amount of data in the buffer

Declaration: `Property BufferSize : Integer`

Visibility: public

Access: Read

Description: `BufferSize` is the actual amount of data in the buffer. This is always less than or equal to the Capacity (71).

See also: TBufStream.Create (70), TBufStream.Buffer (70), TBufStream.BufferPos (71), TBufStream.Capacity (71)

5.5 TReadBufStream

5.5.1 Description

`TReadBufStream` is a read-only buffered stream. It implements the needed methods to read data from the buffer and fill the buffer with additional data when needed.

The stream provides limited forward-seek possibilities.

5.5.2 Method overview

Page	Property	Description
72	Read	Reads data from the stream
72	Seek	Set location in the buffer

5.5.3 TReadBufStream.Seek

Synopsis: Set location in the buffer

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` sets the location in the buffer. Currently, only a forward seek is allowed. It is emulated by reading and discarding data. For an explanation of the parameters, see TStream.Seek" (**??**)

The seek method needs enhancement to enable it to do a full-featured seek. This may be implemented in a future release of Free Pascal.

Errors: In case an illegal seek operation is attempted, an exception is raised.

See also: TWriteBufStream.Seek (73), TReadBufStream.Read (72), TReadBufStream.Write (72)

5.5.4 TReadBufStream.Read

Synopsis: Reads data from the stream

Declaration: `function Read(var ABuffer;ACount: LongInt) : Integer;  Override`

Visibility: public

Description: `Read` reads at most `ACount` bytes from the stream and places them in `Buffer`. The number of actually read bytes is returned.

`TReadBufStream` first reads whatever data is still available in the buffer, and then refills the buffer, after which it continues to read data from the buffer. This is repeated untill `ACount` bytes are read, or no more data is available.

See also: TReadBufStream.Seek (72), TReadBufStream.Read (72)

5.6 TWriteBufStream

5.6.1 Description

`TWriteBufStream` is a write-only buffered stream. It implements the needed methods to write data to the buffer and flush the buffer (i.e., write its contents to the source stream) when needed.

5.6.2 Method overview

Page	Property	Description
73	Destroy	Remove the `TWriteBufStream` instance from memory
73	Seek	Set stream position.
73	Write	Write data to the stream

5.6.3 TWriteBufStream.Destroy

Synopsis: Remove the `TWriteBufStream` instance from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` flushes the buffer and then calls the inherited Destroy (70).

Errors: If an error occurs during flushing of the buffer, an exception may be raised.

See also: TBufStream.Create (70), TBufStream.Destroy (70)

5.6.4 TWriteBufStream.Seek

Synopsis: Set stream position.

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` always raises an `EStreamError` exception, except when the seek operation would not alter the current position.

A later implementation may perform a proper seek operation by flushing the buffer and doing a seek on the source stream.

Errors:

See also: TWriteBufStream.Write (73), TWriteBufStream.Read (72), TReadBufStream.Seek (72)

5.6.5 TWriteBufStream.Write

Synopsis: Write data to the stream

Declaration: `function Write(const ABuffer;ACount: LongInt) : Integer;  Override`

Visibility: public

Description: `Write` writes at most `ACount` bytes from `ABuffer` to the stream. The data is written to the internal buffer first. As soon as the internal buffer is full, it is flushed to the destination stream, and the internal buffer is filled again. This process continues till all data is written (or an error occurs).

Errors: An exception may occur if the destination stream has problems writing.

See also: TWriteBufStream.Seek (73), TWriteBufStream.Read (72), TReadBufStream.Write (72)

Chapter 6

Reference for unit 'CacheCls'

6.1 Used units

Table 6.1: Used units by unit 'CacheCls'

Name	Page
sysutils	??

6.2 Overview

The CacheCls unit implements a caching class: similar to a hash class, it can be used to cache data, associated with string values (keys). The class is calles `TCache`

6.3 Constants, types and variables

6.3.1 Resource strings

```
SInvalidIndex = 'Invalid index %i'
```

Message shown when an invalid index is passed.

6.3.2 Types

```
PCacheSlot = ^TCacheSlot
```

Pointer to TCacheSlot (75) record.

```
PCacheSlotArray = ^TCacheSlotArray
```

Pointer to TCacheSlotArray (75) array

```
TCacheSlot = record
```

```
  Prev : PCacheSlot;
  Next : PCacheSlot;
  Data : Pointer;
  Index : Integer;
end
```

TCacheSlot is internally used by the TCache (75) class. It represents 1 element in the linked list.

```
TCacheSlotArray = Array[0..MaxIntdivSizeOf(TCacheSlot)-1] of TCacheSlot
```

TCacheSlotArray is an array of TCacheSlot items. Do not use TCacheSlotArray directly, instead, use PCacheSlotArray (74) and allocate memory dynamically.

```
TOnFreeSlot = procedure(ACache: TCache;SlotIndex: Integer) of object
```

TOnFreeSlot is a callback prototype used when not enough slots are free, and a slot must be freed.

```
TOnIsDataEqual = function(ACache: TCache;AData1: Pointer;
                          AData2: Pointer) : Boolean of object
```

TOnIsDataEqual is a callback prototype; It is used by the TCache.Add (76) call to determine whether the item to be added is a new item or not. The function returns True if the 2 data pointers AData1 and AData2 should be considered equal, or False when they are not.

For most purposes, comparing the pointers will be enough, but if the pointers are ansistrings, then the contents should be compared.

6.4 ECacheError

6.4.1 Description

Exception class used in the cachecls unit.

6.5 TCache

6.5.1 Description

TCache implements a cache class: it is a list-like class, but which uses a counting mechanism, and keeps a Most-Recent-Used list; this list represents the 'cache'. The list is internally kept as a doubly-linked list.

The Data (78) property offers indexed access to the array of items. When accessing the array through this property, the MRUSlot (78) property is updated.

6.5.2 Method overview

Page	Property	Description
76	Add	Add a data element to the list.
77	AddNew	Add a new item to the list.
76	Create	Create a new cache class.
76	Destroy	Free the `TCache` class from memory
77	FindSlot	Find data pointer in the list
77	IndexOf	Return index of a data pointer in the list.
78	Remove	Remove a data item from the list.

6.5.3 Property overview

Page	Property	Access	Description
78	Data	rw	Indexed access to data items
79	LRUSlot	r	Last used item
78	MRUSlot	rw	Most recent item slot.
80	OnFreeSlot	rw	Event called when a slot is freed
79	OnIsDataEqual	rw	Event to compare 2 items.
79	SlotCount	rw	Number of slots in the list
79	Slots	r	Indexed array to the slots

6.5.4 TCache.Create

Synopsis: Create a new cache class.

Declaration: `constructor Create(ASlotCount: Integer)`

Visibility: public

Description: `Create` instantiates a new instance of `TCache`. It allocates room for `ASlotCount` entries in the list. The number of slots can be increased later.

See also: TCache.SlotCount (79)

6.5.5 TCache.Destroy

Synopsis: Free the `TCache` class from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the array for the elements, and calls the inherited `Destroy`. The elements in the array are not freed by this action.

See also: TCache.Create (76)

6.5.6 TCache.Add

Synopsis: Add a data element to the list.

Declaration: `function Add(AData: Pointer) : Integer`

Visibility: public

Description: `Add` checks whether `AData` is already in the list. If so, the item is added to the top of the MRU list. If the item is not yet in the list, then the item is added to the list and placed at the top of the MRU list using the AddNew (77) call.

The function returns the index at which the item was added.

If the maximum number of slots is reached, and a new item is being added, the least used item is dropped from the list.

See also: TCache.AddNew (77), TCache.FindSlot (77), TCache.IndexOf (77), TCache.Data (78), TCache.MRUSlot (78)

6.5.7 TCache.AddNew

Synopsis: Add a new item to the list.

Declaration: `function AddNew(AData: Pointer) : Integer`

Visibility: public

Description: `AddNew` adds a new item to the list: in difference with the Add (76) call, no checking is performed to see whether the item is already in the list.

The function returns the index at which the item was added.

If the maximum number of slots is reached, and a new item is being added, the least used item is dropped from the list.

See also: TCache.Add (76), TCache.FindSlot (77), TCache.IndexOf (77), TCache.Data (78), TCache.MRUSlot (78)

6.5.8 TCache.FindSlot

Synopsis: Find data pointer in the list

Declaration: `function FindSlot(AData: Pointer) : PCacheSlot`

Visibility: public

Description: `FindSlot` checks all items in the list, and returns the slot which contains a data pointer that matches the pointer `AData`.

If no item with data pointer that matches `AData` is found, `Nil` is returned.

For this function to work correctly, the OnIsDataEqual (79) event must be set.

Errors: If `OnIsDataEqual` is not set, an exception wil be raised.

See also: TCache.IndexOf (77), TCache.Add (76), TCache.OnIsDataEqual (79)

6.5.9 TCache.IndexOf

Synopsis: Return index of a data pointer in the list.

Declaration: `function IndexOf(AData: Pointer) : Integer`

Visibility: public

Description: `IndexOF` searches in the list for a slot with data pointer that matches `AData` and returns the index of the slot.

If no item with data pointer that matches `AData` is found, `-1` is returned.

For this function to work correctly, the OnIsDataEqual (79) event must be set.

Errors: If `OnIsDataEqual` is not set, an exception wil be raised.

See also: TCache.FindSlot (77), TCache.Add (76), TCache.OnIsDataEqual (79)

6.5.10 TCache.Remove

Synopsis: Remove a data item from the list.

Declaration: `procedure Remove(AData: Pointer)`

Visibility: public

Description: `Remove` searches the slot which matches `AData` and if it is found, sets the data pointer to `Nil`, thus effectively removing the pointer from the list.

Errors: None.

See also: TCache.FindSlot (77)

6.5.11 TCache.Data

Synopsis: Indexed access to data items

Declaration: `Property Data[SlotIndex: Integer]: Pointer`

Visibility: public

Access: Read,Write

Description: `Data` offers index-based access to the data pointers in the cache. By accessing an item in the list in this manner, the item is moved to the front of the MRU list, i.e. MRUSlot (78) will point to the accessed item. The access is both read and write.

The index is zero-based and can maximally be SlotCount-1 (79). Providing an invalid index will result in an exception.

See also: TCache.MRUSlot (78)

6.5.12 TCache.MRUSlot

Synopsis: Most recent item slot.

Declaration: `Property MRUSlot : PCacheSlot`

Visibility: public

Access: Read,Write

Description: `MRUSlot` points to the most recent used slot. The most recent used slot is updated when the list is accessed through the Data (78) property, or when an item is added to the list with Add (76) or AddNew (77)

See also: TCache.Add (76), TCache.AddNew (77), TCache.Data (78), TCache.LRUSlot (79)

6.5.13 TCache.LRUSlot

Synopsis: Last used item

Declaration: `Property LRUSlot : PCacheSlot`

Visibility: public

Access: Read

Description: `LRUSlot` points to the least recent used slot. It is the last item in the chain of slots.

See also: TCache.Add (76), TCache.AddNew (77), TCache.Data (78), TCache.MRUSlot (78)

6.5.14 TCache.SlotCount

Synopsis: Number of slots in the list

Declaration: `Property SlotCount : Integer`

Visibility: public

Access: Read,Write

Description: `SlotCount` is the number of slots in the list. Its initial value is set when the `TCache` instance is created, but this can be changed at any time. If items are added to the list and the list is full, then the number of slots is not increased, but the least used item is dropped from the list. In that case OnFreeSlot (80) is called.

See also: TCache.Create (76), TCache.Data (78), TCache.Slots (79)

6.5.15 TCache.Slots

Synopsis: Indexed array to the slots

Declaration: `Property Slots[SlotIndex: Integer]: PCacheSlot`

Visibility: public

Access: Read

Description: `Slots` provides index-based access to the `TCacheSlot` records in the list. Accessing the records directly does not change their position in the MRU list.

The index is zero-based and can maximally be SlotCount-1 (79). Providing an invalid index will result in an exception.

See also: TCache.Data (78), TCache.SlotCount (79)

6.5.16 TCache.OnIsDataEqual

Synopsis: Event to compare 2 items.

Declaration: `Property OnIsDataEqual : TOnIsDataEqual`

Visibility: public

Access: Read,Write

Description: `OnIsDataEqual` is used by FindSlot (77) and IndexOf (77) to compare items when looking for a particular item. These functions are called by the Add (76) method. Failing to set this event will result in an exception. The function should return `True` if the 2 data pointers should be considered equal.

See also: TCache.FindSlot (77), TCache.IndexOf (77), TCache.Add (76)

6.5.17 TCache.OnFreeSlot

Synopsis: Event called when a slot is freed

Declaration: `Property OnFreeSlot : TOnFreeSlot`

Visibility: public

Access: Read,Write

Description: `OnFreeSlot` is called when an item needs to be freed, i.e. when a new item is added to a full list, and the least recent used item needs to be dropped from the list.

The cache class instance and the index of the item to be removed are passed to the callback.

See also: TCache.Add (76), TCache.AddNew (77), TCache.SlotCount (79)

Chapter 7

Reference for unit 'contnrs'

7.1 Used units

Table 7.1: Used units by unit 'contnrs'

Name	Page
Classes	**??**
sysutils	**??**

7.2 Overview

The contnrs unit implements various general-purpose classes:

Object lists lists that manage objects instead of pointers, and which automatically dispose of the objects.

Component lists lists that manage components instead of pointers, and which automatically dispose the components.

Class lists lists that manage class pointers instead of pointers.

Stacks Stack classes to push/pop pointers or objects

Queues Classes to manage a FIFO list of pointers or objects

Hash lists General-purpose Hash lists.

7.3 Constants, types and variables

7.3.1 Constants

```
MaxHashListSize = Maxint div 16
```

`MaxHashListSize` is the maximum number of elements a hash list can contain.

```
MaxHashStrSize = Maxint
```

MaxHashStrSize is the maximum amount of data for the key string values. The key strings are kept in a continuous memory area. This constant determines the maximum size of this memory area.

```
MaxHashTableSize = Maxint div 4
```

MaxHashTableSize is the maximum number of elements in the hash.

```
MaxItemsPerHash = 3
```

MaxItemsPerHash is the threshold above which the hash is expanded. If the number of elements in a hash bucket becomes larger than this value, the hash size is increased.

7.3.2 Types

```
PBucket = ^TBucket
```

Pointer to TBucket (82)" type.

```
PHashItem = ^THashItem
```

PHashItem is a pointer type, pointing to the THashItem (84) record.

```
PHashItemList = ^THashItemList
```

PHashItemList is a pointer to the THashItemList (84). It's used in the TFPHashList (101) as a pointer to the memory area containing the hash item records.

```
PHashTable = ^THashTable
```

PHashTable is a pointer to the THashTable (84). It's used in the TFPHashList (101) as a pointer to the memory area containing the hash values.

```
TBucket = record
  Count : Integer;
  Items : TBucketItemArray;
end
```

TBucket describes 1 bucket in the TCustomBucketList (92) class. It is a container for TBucketItem (83) records. It should never be used directly.

```
TBucketArray = Array of TBucket
```

Array of TBucket (82) records.

```
TBucketItem = record
  Item : Pointer;
  Data : Pointer;
end
```

TBucketItem is a record used for internal use in TCustomBucketList (92). It should not be necessary to use it directly.

```
TBucketItemArray = Array of TBucketItem
```

Array of TBucketItem records

```
TBucketListSizes = (bl2,bl4,bl8,bl16,bl32,bl64,bl128,bl256)
```

Table 7.2: Enumeration values for type TBucketListSizes

Value	Explanation
bl128	List with 128 buckets
bl16	List with 16 buckets
bl2	List with 2 buckets
bl256	List with 256 buckets
bl32	List with 32 buckets
bl4	List with 4 buckets
bl64	List with 64 buckets
bl8	List with 8 buckets

TBucketListSizes is used to set the bucket list size: It specified the number of buckets created by TBucketList (85).

```
TBucketProc = procedure(AInfo: Pointer;AItem: Pointer;AData: Pointer;
                        out AContinue: Boolean)
```

TBucketProc is the prototype for the #TCustomBucketList.Foreach (**??**) call. It is the plain procedural form. The Continue parameter can be set to False to indicate that the Foreach call should stop the iteration.

For a procedure of object (a method) callback, see the TBucketProcObject (83) prototype.

```
TBucketProcObject = procedure(AItem: Pointer;AData: Pointer;
                              out AContinue: Boolean) of object
```

TBucketProcObject is the prototype for the #TCustomBucketList.Foreach (**??**) call. It is the method (procedure of object) form. The Continue parameter can be set to False to indicate that the Foreach call should stop the iteration.

For a plain procedural callback, see the TBucketProc (83) prototype.

```
TDataIteratorMethod = procedure(Item: Pointer;const Key: String;
                                var Continue: Boolean) of object
```

TDataIteratorMethod is a callback prototype for the TDataHashTable.Iterate (81) method. It is called for each data pointer in the hash list, passing the key (key) and data pointer (item) for each item in the list. If Continue is set to false, the iteration stops.

```
THashFunction = function(const S: String;const TableSize: LongWord)
                         : LongWord
```

THashFunction is the prototype for a hash calculation function. It should calculate a hash of string S, where the hash table size is TableSize. The return value should be the hash value.

```
THashItem = record
  HashValue : LongWord;
  StrIndex : Integer;
  NextIndex : Integer;
  Data : Pointer;
end
```

THashItem is used internally in the hash list. It should never be used directly.

```
THashItemList = Array[0..MaxHashListSize-1] of THashItem
```

THashItemList is an array type, primarily used to be able to define the PHashItemList (82) type. It's used in the TFPHashList (101) class.

```
THashTable = Array[0..MaxHashTableSize-1] of Integer
```

THashTable defines an array of integers, used to hold hash values. It's mainly used to define the PHashTable (82) class.

```
THTCustomNodeClass = Class of THTCustomNode
```

THTCustomNodeClass is used by THTCustomHashTable (81) to decide which class should be created for elements in the list.

```
THTNode = THTDataNode
```

THTNode is provided for backwards compatibility.

```
TIteratorMethod = TDataIteratorMethod
```

TIteratorMethod is used in an internal TFPHashTable (81) method.

```
TObjectIteratorMethod = procedure(Item: TObject;const Key: String;
                                  var Continue: Boolean) of object
```

TObjectIteratorMethod is the iterator callback prototype. It is used to iterate over all items in the hash table, and is called with each key value (Key) and associated object (Item). If Continue is set to false, the iteration stops.

```
TObjectListCallback = procedure(data: TObject;arg: pointer) of object
```

TObjectListCallback is used as the prototype for the TFPObjectList.ForEachCall (125) link call when a method should be called. The Data argument will contain each of the objects in the list in turn, and the Data argument will contain the data passed to the ForEachCall call.

```
TObjectListStaticCallback = procedure(data: TObject;arg: pointer)
```

TObjectListCallback is used as the prototype for the TFPObjectList.ForEachCall (125) link call when a plain procedure should be called. The Data argument will contain each of the objects in the list in turn, and the Data argument will contain the data passed to the ForEachCall call.

```
TStringIteratorMethod = procedure(Item: String;const Key: String;
                                  var Continue: Boolean) of object
```

TStringIteratorMethod is the callback prototype for the Iterate (94) method. It is called for each element in the hash table, with the string. If Continue is set to false, the iteration stops.

7.4 Procedures and functions

7.4.1 RSHash

Synopsis: Standard hash value calculating function.

Declaration: `function RSHash(const S: String;const TableSize: LongWord) : LongWord`

Visibility: default

Description: RSHash is the standard hash calculating function used in the TFPCustomHashTable (94) hash class. It's Robert Sedgwick's "Algorithms in C" hash function.

Errors: None.

See also: TFPCustomHashTable (94)

7.5 EDuplicate

7.5.1 Description

Exception raised when a key is stored twice in a hash table.

7.6 EKeyNotFound

7.6.1 Description

Exception raised when a key is not found.

7.7 TBucketList

7.7.1 Description

TBucketList is a descendent of TCustomBucketList which allows to specify a bucket count which is a multiple of 2, up to 256 buckets. The size is passed to the constructor and cannot be changed in the lifetime of the bucket list instance.

The buckets for an item is determined by looking at the last bits of the item pointer: For 2 buckets, the last bit is examined, for 4 buckets, the last 2 bits are taken and so on. The algorithm takes into account the average granularity (4) of heap pointers.

7.7.2 Method overview

Page	Property	Description
86	Create	Create a new `TBucketList` instance.

7.7.3 TBucketList.Create

Synopsis: Create a new `TBucketList` instance.

Declaration: `constructor Create(ABuckets: TBucketListSizes)`

Visibility: public

Description: `Create` instantiates a new bucketlist instance with a number of buckets determined by `ABuckets`. After creation, the number of buckets can no longer be changed.

Errors: If not enough memory is available to create the instance, an exception may be raised.

See also: TBucketListSizes (83)

7.8 TClassList

7.8.1 Description

`TClassList` is a Tlist (**??**) descendent which stores class references instead of pointers. It introduces no new behaviour other than ensuring all stored pointers are class pointers.

The `OwnsObjects` property as found in `TComponentList` and `TObjectList` is not implemented as there are no actual instances.

7.8.2 Method overview

Page	Property	Description
86	Add	Add a new class pointer to the list.
87	Extract	Extract a class pointer from the list.
88	First	Return first non-nil class pointer
87	IndexOf	Search for a class pointer in the list.
88	Insert	Insert a new class pointer in the list.
88	Last	Return last non-`Nil` class pointer
87	Remove	Remove a class pointer from the list.

7.8.3 Property overview

Page	Property	Access	Description
88	Items	rw	Index based access to class pointers.

7.8.4 TClassList.Add

Synopsis: Add a new class pointer to the list.

Declaration: `function Add(AClass: TClass) : Integer`

Visibility: public

Description: `Add` adds `AClass` to the list, and returns the position at which it was added. It simply overrides the TList (**??**) bevahiour, and introduces no new functionality.

Errors: If not enough memory is available to expand the list, an exception may be raised.

See also: TClassList.Extract (87), #rtl.classes.tlist.add (**??**)

7.8.5 TClassList.Extract

Synopsis: Extract a class pointer from the list.

Declaration: `function Extract(Item: TClass) : TClass`

Visibility: public

Description: `Extract` extracts a class pointer `Item` from the list, if it is present in the list. It returns the extracted class pointer, or `Nil` if the class pointer was not present in the list. It simply overrides the implementation in `TList` so it accepts a class pointer instead of a simple pointer. No new behaviour is introduced.

Errors: None.

See also: TClassList.Remove (87), #rtl.classes.Tlist.Extract (**??**)

7.8.6 TClassList.Remove

Synopsis: Remove a class pointer from the list.

Declaration: `function Remove(AClass: TClass) : Integer`

Visibility: public

Description: `Remove` removes a class pointer `Item` from the list, if it is present in the list. It returns the index of the removed class pointer, or `-1` if the class pointer was not present in the list. It simply overrides the implementation in `TList` so it accepts a class pointer instead of a simple pointer. No new behaviour is introduced.

Errors: None.

See also: TClassList.Extract (87), #rtl.classes.Tlist.Remove (**??**)

7.8.7 TClassList.IndexOf

Synopsis: Search for a class pointer in the list.

Declaration: `function IndexOf(AClass: TClass) : Integer`

Visibility: public

Description: `IndexOf` searches for `AClass` in the list, and returns it's position if it was found, or -1 if it was not found in the list.

Errors: None.

See also: #rtl.classes.tlist.indexof (**??**)

7.8.8 TClassList.First

Synopsis: Return first non-nil class pointer

Declaration: `function First : TClass`

Visibility: public

Description: `First` returns a reference to the first non-`Nil` class pointer in the list. If no non-`Nil` element is found, `Nil` is returned.

Errors: None.

See also: TClassList.Last (88), TClassList.Pack (86)

7.8.9 TClassList.Last

Synopsis: Return last non-`Nil` class pointer

Declaration: `function Last : TClass`

Visibility: public

Description: `Last` returns a reference to the last non-`Nil` class pointer in the list. If no non-`Nil` element is found, `Nil` is returned.

Errors: None.

See also: TClassList.First (88), TClassList.Pack (86)

7.8.10 TClassList.Insert

Synopsis: Insert a new class pointer in the list.

Declaration: `procedure Insert(Index: Integer;AClass: TClass)`

Visibility: public

Description: `Insert` inserts a class pointer in the list at position `Index`. It simply overrides the parent implementation so it only accepts class pointers. It introduces no new behaviour.

Errors: None.

See also: #rtl.classes.TList.Insert (**??**), TClassList.Add (86), TClassList.Remove (87)

7.8.11 TClassList.Items

Synopsis: Index based access to class pointers.

Declaration: `Property Items[Index: Integer]: TClass; default`

Visibility: public

Access: Read,Write

Description: `Items` provides index-based access to the class pointers in the list. `TClassList` overrides the default `Items` implementation of `TList` so it returns class pointers instead of pointers.

See also: #rtl.classes.TList.Items (**??**), #rtl.classes.TList.Count (**??**)

7.9 TComponentList

7.9.1 Description

`TComponentList` is a TObjectList (132) descendent which has as the default array property TComponents (**??**) instead of objects. It overrides some methods so only components can be added.

In difference with TObjectList (132), `TComponentList` removes any `TComponent` from the list if the `TComponent` instance was freed externally. It uses the `FreeNotification` mechanism for this.

7.9.2 Method overview

Page	Property	Description
89	Add	Add a component to the list.
89	Destroy	Destroys the instance
90	Extract	Remove a component from the list without destroying it.
91	First	First non-nil instance in the list.
90	IndexOf	Search for an instance in the list
91	Insert	Insert a new component in the list
91	Last	Last non-nil instance in the list.
90	Remove	Remove a component from the list, possibly destroying it.

7.9.3 Property overview

Page	Property	Access	Description
91	Items	rw	Index-based access to the elements in the list.

7.9.4 TComponentList.Destroy

Synopsis: Destroys the instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` unhooks the free notification handler and then calls the inherited destroy to clean up the `TComponentList` instance.

Errors: None.

See also: TObjectList (132), #rtl.classes.TComponent (**??**)

7.9.5 TComponentList.Add

Synopsis: Add a component to the list.

Declaration: `function Add(AComponent: TComponent) : Integer`

Visibility: public

Description: `Add` overrides the `Add` operation of it's ancestors, so it only accepts `TComponent` instances. It introduces no new behaviour.

The function returns the index at which the component was added.

Errors: If not enough memory is available to expand the list, an exception may be raised.

See also: TObectList.Add (81)

7.9.6 TComponentList.Extract

Synopsis: Remove a component from the list without destroying it.

Declaration: `function Extract(Item: TComponent) : TComponent`

Visibility: public

Description: `Extract` removes a component (`Item`) from the list, without destroying it. It overrides the implementation of TObjectList (132) so only `TComponent` descendents can be extracted. It introduces no new behaviour.

`Extract` returns the instance that was extracted, or `Nil` if no instance was found.

See also: TComponentList.Remove (90), TObjectList.Extract (133)

7.9.7 TComponentList.Remove

Synopsis: Remove a component from the list, possibly destroying it.

Declaration: `function Remove(AComponent: TComponent) : Integer`

Visibility: public

Description: `Remove` removes `item` from the list, and if the list owns it's items, it also destroys it. It returns the index of the item that was removed, or -1 if no item was removed.

`Remove` simply overrides the implementation in TObjectList (132) so it only accepts `TComponent` descendents. It introduces no new behaviour.

Errors: None.

See also: TComponentList.Extract (90), TObjectList.Remove (134)

7.9.8 TComponentList.IndexOf

Synopsis: Search for an instance in the list

Declaration: `function IndexOf(AComponent: TComponent) : Integer`

Visibility: public

Description: `IndexOf` searches for an instance in the list and returns it's position in the list. The position is zero-based. If no instance is found, -1 is returned.

`IndexOf` just overrides the implementation of the parent class so it accepts only `TComponent` instances. It introduces no new behaviour.

Errors: None.

See also: TObjectList.IndexOf (134)

7.9.9 TComponentList.First

Synopsis: First non-nil instance in the list.

Declaration: `function First : TComponent`

Visibility: public

Description: `First` overrides the implementation of it's ancestors to return the first non-nil instance of `TComponent` in the list. If no non-nil instance is found, `Nil` is returned.

Errors: None.

See also: TComponentList.Last (91), TObjectList.First (135)

7.9.10 TComponentList.Last

Synopsis: Last non-nil instance in the list.

Declaration: `function Last : TComponent`

Visibility: public

Description: `Last` overrides the implementation of it's ancestors to return the last non-nil instance of `TComponent` in the list. If no non-nil instance is found, `Nil` is returned.

Errors: None.

See also: TComponentList.First (91), TObjectList.Last (135)

7.9.11 TComponentList.Insert

Synopsis: Insert a new component in the list

Declaration: `procedure Insert(Index: Integer;AComponent: TComponent)`

Visibility: public

Description: `Insert` inserts a `TComponent` instance (`AComponent`) in the list at position `Index`. It simply overrides the parent implementation so it only accepts `TComponent` instances. It introduces no new behaviour.

Errors: None.

See also: TObjectList.Insert (135), TComponentList.Add (89), TComponentList.Remove (90)

7.9.12 TComponentList.Items

Synopsis: Index-based access to the elements in the list.

Declaration: `Property Items[Index: Integer]: TComponent; default`

Visibility: public

Access: Read,Write

Description: `Items` provides access to the components in the list using an index. It simply overrides the default property of the parent classes so it returns/accepts `TComponent` instances only. Note that the index is zero based.

See also: TObjectList.Items (136)

7.10 TCustomBucketList

7.10.1 Description

`TCustomBucketList` is an associative list using buckets for storage. It scales better than a regular TList (**??**) list class, escpecially when an item must be searched in the list.

Since the list associates a data pointer with each item pointer, it follows that each item pointer must be unique, and can be added to the list only once.

The `TCustomBucketList` class does not determine the number of buckets or the bucket hash mechanism, this must be done by descendent classes such as TBucketList (85). `TCustomBucketList` only takes care of storage and retrieval of items in the various buckets.

Because `TCustomBucketList` is an abstract class - it does not determine the number of buckets - one should never instantiate an instance of `TCustomBucketList`, but always use a descendent class such as TCustomBucketList (92).

7.10.2 Method overview

Page	Property	Description
93	Add	Add an item to the list
93	Assign	Assign one bucket list to another
92	Clear	Clear the list
92	Destroy	Frees the bucketlist from memory
93	Exists	Check if an item exists in the list.
93	Find	Find an item in the list
94	ForEach	Loop over all items.
94	Remove	Remove an item from the list.

7.10.3 Property overview

Page	Property	Access	Description
94	Data	rw	Associative array for data pointers

7.10.4 TCustomBucketList.Destroy

Synopsis: Frees the bucketlist from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Detsroy` frees all storage for the buckets from memory. The items themselves are not freed from memory.

7.10.5 TCustomBucketList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` clears the list. The items and their data themselves are not disposed of, this must be done separately. Clear only removes all references to the items from the list.

Errors: None.

See also: TCustomBucketList.Add (93)

7.10.6 TCustomBucketList.Add

Synopsis: Add an item to the list

Declaration: `function Add(AItem: Pointer;AData: Pointer) : Pointer`

Visibility: public

Description: `Add` adds `AItem` with it's associated `AData` to the list and returns `AData`.

Errors: If `AItem` is already in the list, an `EListError` exception will be raised.

See also: TCustomBucketList.Exists (93), TCustomBucketList.Clear (92)

7.10.7 TCustomBucketList.Assign

Synopsis: Assign one bucket list to another

Declaration: `procedure Assign(AList: TCustomBucketList)`

Visibility: public

Description: `Assign` is implemented by `TCustomBucketList` to copy the contents of another bucket list to the bucket list. It clears the contents prior to the copy operation.

See also: TCustomBucketList.Add (93), TCustomBucketList.Clear (92)

7.10.8 TCustomBucketList.Exists

Synopsis: Check if an item exists in the list.

Declaration: `function Exists(AItem: Pointer) : Boolean`

Visibility: public

Description: `Exists` searches the list and returns `True` if the `AItem` is already present in the list. If the item is not yet in the list, `False` is returne5Ad.

If the data pointer associated with `AItem` is also needed, then it is better to use Find (93).

See also: TCustomBucketList.Find (93)

7.10.9 TCustomBucketList.Find

Synopsis: Find an item in the list

Declaration: `function Find(AItem: Pointer;out AData: Pointer) : Boolean`

Visibility: public

Description: `Find` searches for `AItem` in the list and returns the data pointer associated with it in `AData` if the item was found. In that case the return value is `True`. If `AItem` is not found in the list, `False` is returned.

See also: TCustomBucketList.Exists (93)

7.10.10 TCustomBucketList.ForEach

Synopsis: Loop over all items.

Declaration:
```
function ForEach(AProc: TBucketProc;AInfo: Pointer) : Boolean
function ForEach(AProc: TBucketProcObject) : Boolean
```

Visibility: public

Description: `Foreach` loops over all items in the list and calls `AProc`, passing it in turn each item in the list.

`AProc` exists in 2 variants: one which is a simple procedure, and one which is a method. In the case of the simple procedure, the `AInfo` argument is passed as well in each call to `AProc`.

The loop stops when all items have been processed, or when the `AContinue` argument of `AProc` contains `False` on return.

The result of the function is `True` if all items were processed, or `False` if the loop was interrupted with a `AContinue` return of `False`.

Errors: None.

See also: TCustomBucketList.Data (94)

7.10.11 TCustomBucketList.Remove

Synopsis: Remove an item from the list.

Declaration:
```
function Remove(AItem: Pointer) : Pointer
```

Visibility: public

Description: `Remove` removes `AItem` from the list, and returns the associated data pointer of the removed item. If the item was not in the list, then `Nil` is returned.

See also: TCustomBucketList.Find (93)

7.10.12 TCustomBucketList.Data

Synopsis: Associative array for data pointers

Declaration:
```
Property Data[AItem: Pointer]: Pointer; default
```

Visibility: public

Access: Read,Write

Description: `Data` provides direct access to the `Data` pointers associated with the `AItem` pointers. If `AItem` is not in the list of pointers, an `EListError` exception will be raised.

See also: TCustomBucketList.Find (93), TCustomBucketList.Exists (93)

7.11 TFPCustomHashTable

7.11.1 Description

`TFPCustomHashTable` is a general-purpose hashing class. It can store string keys and pointers associated with these strings. The hash mechanism is configurable and can be optionally be specified

when a new instance of the class is created; A default hash mechanism is implemented in RSHash (85).

A `TFPHasList` should be used when fast lookup of data based on some key is required. The other container objects only offer linear search methods, while the hash list offers faster search mechanisms.

7.11.2 Method overview

Page	Property	Description
96	ChangeTableSize	Change the table size of the hash table.
96	Clear	Clear the hash table.
95	Create	Instantiate a new `TFPCustomHashTable` instance using the default hash mechanism
95	CreateWith	Instantiate a new `TFPCustomHashTable` instance with given algorithm and size
97	Delete	Delete a key from the hash list.
96	Destroy	Free the hash table.
97	Find	Search for an item with a certain key value.
97	IsEmpty	Check if the hash table is empty.

7.11.3 Property overview

Page	Property	Access	Description
99	AVGChainLen	r	Average chain length
98	Count	r	Number of items in the hash table.
100	Density	r	Number of filled slots
97	HashFunction	rw	Hash function currently in use
98	HashTable	r	Hash table instance
98	HashTableSize	rw	Size of the hash table
99	LoadFactor	r	Fraction of count versus size
99	MaxChainLength	r	Maximum chain length
100	NumberOfCollisions	r	Number of extra items
99	VoidSlots	r	Number of empty slots in the hash table.

7.11.4 TFPCustomHashTable.Create

Synopsis: Instantiate a new `TFPCustomHashTable` instance using the default hash mechanism

Declaration: `constructor Create`

Visibility: public

Description: `Create` creates a new instance of `TFPCustomHashTable` with hash size 196613 and hash algorithm RSHash (85)

Errors: If no memory is available, an exception may be raised.

See also: TFPCustomHashTable.CreateWith (95)

7.11.5 TFPCustomHashTable.CreateWith

Synopsis: Instantiate a new `TFPCustomHashTable` instance with given algorithm and size

Declaration: `constructor CreateWith(AHashTableSize: LongWord; aHashFunc: THashFunction)`

Visibility: public

Description: `CreateWith` creates a new instance of `TFPCustomHashTable` with hash size `AHashTableSize` and hash calculating algorithm `aHashFunc`.

Errors: If no memory is available, an exception may be raised.

See also: TFPCustomHashTable.Create (95)

7.11.6 TFPCustomHashTable.Destroy

Synopsis: Free the hash table.

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` removes the hash table from memory. If any data was associated with the keys in the hash table, then this data is not freed. This must be done by the programmer.

Errors: None.

See also: TFPCustomHashTable.Destroy (96), TFPCustomHashTable.Create (95), TFPCustomHashTable.CreateWith (95), THTCustomNode.Data (128)

7.11.7 TFPCustomHashTable.ChangeTableSize

Synopsis: Change the table size of the hash table.

Declaration: `procedure ChangeTableSize(const ANewSize: LongWord); Virtual`

Visibility: public

Description: `ChangeTableSize` changes the size of the hash table: it recomputes the hash value for all of the keys in the table, so this is an expensive operation.

Errors: If no memory is available, an exception may be raised.

See also: TFPCustomHashTable.HashTableSize (98)

7.11.8 TFPCustomHashTable.Clear

Synopsis: Clear the hash table.

Declaration: `procedure Clear; Virtual`

Visibility: public

Description: `Clear` removes all keys and their associated data from the hash table. The data itself is not freed from memory, this should be done by the programmer.

Errors: None.

See also: TFPCustomHashTable.Destroy (96)

7.11.9 TFPCustomHashTable.Delete

Synopsis: Delete a key from the hash list.

Declaration: `procedure Delete(const aKey: String);  Virtual`

Visibility: public

Description: `Delete` deletes all keys with value `AKey` from the hash table. It does not free the data associated with key. If `AKey` is not in the list, nothing is removed.

Errors: None.

See also: TFPCustomHashTable.Find (97), TFPCustomHashTable.Add (94)

7.11.10 TFPCustomHashTable.Find

Synopsis: Search for an item with a certain key value.

Declaration: `function Find(const aKey: String) : THTCustomNode`

Visibility: public

Description: `Find` searches for the THTCustomNode (128) instance with key value equal to `Akey` and if it finds it, it returns the instance. If no matching value is found, `Nil` is returned.

Note that the instance returned by this function cannot be freed; If it should be removed from the hash table, the Delete (97) method should be used instead.

Errors: None.

See also: TFPCustomHashTable.Add (94), TFPCustomHashTable.Delete (97)

7.11.11 TFPCustomHashTable.IsEmpty

Synopsis: Check if the hash table is empty.

Declaration: `function IsEmpty : Boolean`

Visibility: public

Description: `IsEmpty` returns `True` if the hash table contains no elements, or `False` if there are still elements in the hash table.

Errors:

See also: TFPCustomHashTable.Count (98), TFPCustomHashTable.HashTableSize (98), TFPCustomHashTable.AVGChainLen (99), TFPCustomHashTable.MaxChainLength (99)

7.11.12 TFPCustomHashTable.HashFunction

Synopsis: Hash function currently in use

Declaration: `Property HashFunction : THashFunction`

Visibility: public

Access: Read,Write

Description: `HashFunction` is the hash function currently in use to calculate hash values from keys. The property can be set, this simply calls SetHashFunction (94). Note that setting the hash function does NOT the hash value of all keys to be recomputed, so changing the value while there are still keys in the table is not a good idea.

See also: TFPCustomHashTable.SetHashFunction (94), TFPCustomHashTable.HashTableSize (98)

7.11.13 TFPCustomHashTable.Count

Synopsis: Number of items in the hash table.

Declaration: `Property Count : LongWord`

Visibility: public

Access: Read

Description: `Count` is the number of items in the hash table.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.HashTableSize (98), TFPCustomHashTable.AVGChainLen (99), TFPCustomHashTable.MaxChainLength (99)

7.11.14 TFPCustomHashTable.HashTableSize

Synopsis: Size of the hash table

Declaration: `Property HashTableSize : LongWord`

Visibility: public

Access: Read,Write

Description: `HashTableSize` is the size of the hash table. It can be set, in which case it will be rounded to the nearest prime number suitable for RSHash.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.AVGChainLen (99), TFPCustomHashTable.MaxChainLength (99), TFPCustomHashTable.VoidSlots (99), TFPCustomHashTable.Density (100)

7.11.15 TFPCustomHashTable.HashTable

Synopsis: Hash table instance

Declaration: `Property HashTable : TFPObjectList`

Visibility: public

Access: Read

Description: `TFPCustomHashTable` is the internal list object (TFPObjectList (120) used for the hash table. Each element in this table is again a TFPObjectList (120) instance or `Nil`.

7.11.16 TFPCustomHashTable.VoidSlots

Synopsis: Number of empty slots in the hash table.

Declaration: `Property VoidSlots : LongWord`

Visibility: public

Access: Read

Description: `VoidSlots` is the number of empty slots in the hash table. Calculating this is an expensive operation.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.AVGChainLen (99), TFPCustomHashTable.MaxChainLength (99), TFPCustomHashTable.LoadFactor (99), TFPCustomHashTable.Density (100), TFPCustomHashTable.NumberOfCollisions (100)

7.11.17 TFPCustomHashTable.LoadFactor

Synopsis: Fraction of count versus size

Declaration: `Property LoadFactor : double`

Visibility: public

Access: Read

Description: `LoadFactor` is the ratio of elements in the table versus table size. Ideally, this should be as small as possible.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.AVGChainLen (99), TFPCustomHashTable.MaxChainLength (99), TFPCustomHashTable.VoidSlots (99), TFPCustomHashTable.Density (100), TFPCustomHashTable.NumberOfCollisions (100)

7.11.18 TFPCustomHashTable.AVGChainLen

Synopsis: Average chain length

Declaration: `Property AVGChainLen : double`

Visibility: public

Access: Read

Description: `AVGChainLen` is the average chain length, i.e. the ratio of elements in the table versus the number of filled slots. Calculating this is an expensive operation.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.LoadFactor (99), TFPCustomHashTable.MaxChainLength (99), TFPCustomHashTable.VoidSlots (99), TFPCustomHashTable.Density (100), TFPCustomHashTable.NumberOfCollisions (100)

7.11.19 TFPCustomHashTable.MaxChainLength

Synopsis: Maximum chain length

Declaration: `Property MaxChainLength : LongWord`

Visibility: public

Access: Read

Description: `MaxChainLength` is the length of the longest chain in the hash table. Calculating this is an expensive operation.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.LoadFactor (99), TFPCustomHashTable.AvgChainLength (94), TFPCustomHashTable.VoidSlots (99), TFPCustomHashTable.Density (100), TFPCustomHashTable.NumberOfCollisions (100)

7.11.20 TFPCustomHashTable.NumberOfCollisions

Synopsis: Number of extra items

Declaration: `Property NumberOfCollisions : LongWord`

Visibility: public

Access: Read

Description: `NumberOfCollisions` is the number of items which are not the first item in a chain. If this number is too big, the hash size may be too small.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.LoadFactor (99), TFPCustomHashTable.AvgChainLength (94), TFPCustomHashTable.VoidSlots (99), TFPCustomHashTable.Density (100)

7.11.21 TFPCustomHashTable.Density

Synopsis: Number of filled slots

Declaration: `Property Density : LongWord`

Visibility: public

Access: Read

Description: `Density` is the number of filled slots in the hash table.

See also: TFPCustomHashTable.IsEmpty (97), TFPCustomHashTable.Count (98), TFPCustomHashTable.LoadFactor (99), TFPCustomHashTable.AvgChainLength (94), TFPCustomHashTable.VoidSlots (99), TFPCustomHashTable.Density (100)

7.12 TFPDataHashTable

7.12.1 Description

`TFPDataHashTable` is a TFPCustomHashTable (94) descendent which stores simple data pointers together with the keys. In case the data associated with the keys are objects, it's better to use TFPObjectHashTable (118), or for string data, TFPStringHashTable (127) is more suitable. The data pointers are exposed with their keys through the Items (101) property.

7.12.2 Method overview

Page	Property	Description
101	Add	Add a data pointer to the list.

7.12.3 Property overview

Page	Property	Access	Description
101	Items	rw	Key-based access to the items in the table

7.12.4 TFPDataHashTable.Add

Synopsis: Add a data pointer to the list.

Declaration: `procedure Add(const aKey: String;AItem: pointer);  Virtual`

Visibility: public

Description: `Add` adds a data pointer (`AItem`) to the list with key `AKey`.

Errors: If `AKey` already exists in the table, an exception is raised.

See also: TFPDataHashTable.Items (101)

7.12.5 TFPDataHashTable.Items

Synopsis: Key-based access to the items in the table

Declaration: `Property Items[index: String]: Pointer; default`

Visibility: public

Access: Read,Write

Description: `Items` provides access to the items in the hash table using their key: the array index `Index` is the key. A key which is not present will result in an `Nil` pointer.

See also: TFPStringHashTable.Add (127)

7.13 TFPHashList

7.13.1 Description

`TFPHashList` implements a fast hash class. The class is built for speed, therefore the key values can be shortstrings only, and the data can only be pointers.

if a base class for an own hash class is wanted, the TFPCustomHashTable (94) class can be used. If a hash class for objects is needed instead of pointers, the TFPHashObjectList (111) class can be used.

7.13.2 Method overview

Page	Property	Description
103	Add	Add a new key/data pair to the list
103	Clear	Clear the list
102	Create	Create a new instance of the hashlist
104	Delete	Delete an item from the list.
102	Destroy	Removes an instance of the hashlist from the heap
104	Error	Raise an error
104	Expand	Expand the list
105	Extract	Extract a pointer from the list
105	Find	Find data associated with key
105	FindIndexOf	Return index of named item.
106	FindWithHash	Find first element with given name and hash value
107	ForEachCall	Call a procedure for each element in the list
104	GetNextCollision	Get next collision number
103	HashOfIndex	Return the hash valye of an item by index
105	IndexOf	Return the index of the data pointer
103	NameOfIndex	Returns the key name of an item by index
106	Pack	Remove nil pointers from the list
106	Remove	Remove first instance of a pointer
106	Rename	Rename a key
107	ShowStatistics	Return some statistics for the list.

7.13.3 Property overview

Page	Property	Access	Description
107	Capacity	rw	Capacity of the list.
107	Count	rw	Current number of elements in the list.
108	Items	rw	Indexed array with pointers
108	List	r	Low-level hash list
108	Strs	r	Low-level memory area with strings.

7.13.4 TFPHashList.Create

Synopsis: Create a new instance of the hashlist

Declaration: `constructor Create`

Visibility: public

Description: `Create` creates a new instance of `TFPHashList` on the heap and sets the hash capacity to 1.

See also: TFPHashList.Destroy (102)

7.13.5 TFPHashList.Destroy

Synopsis: Removes an instance of the hashlist from the heap

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the memory structures maintained by the hashlist and removes the `TFPHashList` instance from the heap.

`Destroy` should not be called directly, it's better to use `Free` or `FreeAndNil` instead.

See also: TFPHashList.Create (102), TFPHashList.Clear (103)

7.13.6 TFPHashList.Add

Synopsis: Add a new key/data pair to the list

Declaration: `function Add(const AName: shortstring;Item: Pointer) : Integer`

Visibility: public

Description: `Add` adds a new data pointer (`Item`) with key `AName` to the list. It returns the position of the item in the list.

Errors: If not enough memory is available to hold the key and data, an exception may be raised. If an item with this name already exists in the list, an exception is raised.

See also: TFPHashList.Extract (105), TFPHashList.Remove (106), TFPHashList.Delete (104)

7.13.7 TFPHashList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` removes all items from the list. It does not free the data items themselves. It frees all memory needed to contain the items.

Errors: None.

See also: TFPHashList.Extract (105), TFPHashList.Remove (106), TFPHashList.Delete (104), TFPHashList.Add (103)

7.13.8 TFPHashList.NameOfIndex

Synopsis: Returns the key name of an item by index

Declaration: `function NameOfIndex(Index: Integer) : ShortString`

Visibility: public

Description: `NameOfIndex` returns the key name of the item at position `Index`.

Errors: If `Index` is out of the valid range, an exception is raised.

See also: TFPHashList.HashOfIndex (103), TFPHashList.Find (105), TFPHashList.FindIndexOf (105), TFPHashList.FindWithHash (106)

7.13.9 TFPHashList.HashOfIndex

Synopsis: Return the hash valye of an item by index

Declaration: `function HashOfIndex(Index: Integer) : LongWord`

Visibility: public

Description: `HashOfIndex` returns the hash value of the item at position `Index`.

Errors: If `Index` is out of the valid range, an exception is raised.

See also: TFPHashList.HashOfName (101), TFPHashList.Find (105), TFPHashList.FindIndexOf (105), TFPHashList.FindWithHash (106)

7.13.10 TFPHashList.GetNextCollision

Synopsis: Get next collision number

Declaration: `function GetNextCollision(Index: Integer) : Integer`

Visibility: public

Description: `GetNextCollision` returns the next collision in hash item `Index`. This is the count of items with the same hash.means that the next it

Errors:

7.13.11 TFPHashList.Delete

Synopsis: Delete an item from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` deletes the item at position `Index`. The data to which it points is not freed from memory.

Errors: TFPHashList.Extract (105)TFPHashList.Remove (106)TFPHashList.Add (103)

7.13.12 TFPHashList.Error

Synopsis: Raise an error

Declaration: `procedure Error(const Msg: String;Data: PtrInt)`

Visibility: public

Description: `Error` raises an `EListError` exception, with message `Msg`. The `Data` pointer is used to format the message.

7.13.13 TFPHashList.Expand

Synopsis: Expand the list

Declaration: `function Expand : TFPHashList`

Visibility: public

Description: `Expand` enlarges the capacity of the list if the maximum capacity was reached. It returns itself.

Errors: If not enough memory is available, an exception may be raised.

See also: TFPHashList.Clear (103)

7.13.14 TFPHashList.Extract

Synopsis: Extract a pointer from the list

Declaration: `function Extract(item: Pointer) : Pointer`

Visibility: public

Description: `Extract` removes the data item from the list, if it is in the list. It returns the pointer if it was removed from the list, `Nil` otherwise.

`Extract` does a linear search, and is not very efficient.

See also: TFPHashList.Delete (104), TFPHashList.Remove (106), TFPHashList.Clear (103)

7.13.15 TFPHashList.IndexOf

Synopsis: Return the index of the data pointer

Declaration: `function IndexOf(Item: Pointer) : Integer`

Visibility: public

Description: `IndexOf` returns the index of the first occurrence of pointer `Item`. If the item is not in the list, -1 is returned.

The performed search is linear, and not very efficient.

See also: TFPHashList.HashOfIndex (103), TFPHashList.NameOfIndex (103), TFPHashList.Find (105), TFPHashList.FindIndexOf (105), TFPHashList.FindWithHash (106)

7.13.16 TFPHashList.Find

Synopsis: Find data associated with key

Declaration: `function Find(const AName: shortstring) : Pointer`

Visibility: public

Description: `Find` searches (using the hash) for the data item associated with item `AName` and returns the data pointer associated with it. If the item is not found, `Nil` is returned. It uses the hash value of the key to perform the search.

See also: TFPHashList.HashOfIndex (103), TFPHashList.NameOfIndex (103), TFPHashList.IndexOf (105), TFPHashList.FindIndexOf (105), TFPHashList.FindWithHash (106)

7.13.17 TFPHashList.FindIndexOf

Synopsis: Return index of named item.

Declaration: `function FindIndexOf(const AName: shortstring) : Integer`

Visibility: public

Description: `FindIndexOf` returns the index of the key `AName`, or -1 if the key does not exist in the list. It uses the hash value to search for the key.

See also: TFPHashList.HashOfIndex (103), TFPHashList.NameOfIndex (103), TFPHashList.IndexOf (105), TFPHashList.Find (105), TFPHashList.FindWithHash (106)

7.13.18 TFPHashList.FindWithHash

Synopsis: Find first element with given name and hash value

Declaration:
```
function FindWithHash(const AName: shortstring;AHash: LongWord)
                      : Pointer
```

Visibility: public

Description: `FindWithHash` searches for the item with key `AName`. It uses the provided hash value `AHash` to perform the search. If the item exists, the data pointer is returned, if not, the result is `Nil`.

See also: TFPHashList.HashOfIndex (103), TFPHashList.NameOfIndex (103), TFPHashList.IndexOf (105), TFPHashList.Find (105), TFPHashList.FindIndexOf (105)

7.13.19 TFPHashList.Rename

Synopsis: Rename a key

Declaration:
```
function Rename(const AOldName: shortstring;const ANewName: shortstring)
                : Integer
```

Visibility: public

Description: `Rename` renames key `AOldname` to `ANewName`. The hash value is recomputed and the item is moved in the list to it's new position.

Errors: If an item with `ANewName` already exists, an exception will be raised.

7.13.20 TFPHashList.Remove

Synopsis: Remove first instance of a pointer

Declaration:
```
function Remove(Item: Pointer) : Integer
```

Visibility: public

Description: `Remove` removes the first occurence of the data pointer `Item` in the list, if it is present. The return value is the removed data pointer, or `Nil` if no data pointer was removed.

See also: TFPHashList.Delete (104), TFPHashList.Clear (103), TFPHashList.Extract (105)

7.13.21 TFPHashList.Pack

Synopsis: Remove nil pointers from the list

Declaration:
```
procedure Pack
```

Visibility: public

Description: `Pack` removes all `Nil` items from the list, and frees all unused memory.

See also: TFPHashList.Clear (103)

7.13.22 TFPHashList.ShowStatistics

Synopsis: Return some statistics for the list.

Declaration:
```
procedure ShowStatistics
```

Visibility: public

Description: `ShowStatistics` prints some information about the hash list to standard output. It prints the following values:

HashSizeSize of the hash table

HashMeanMean hash value

HashStdDevStandard deviation of hash values

ListSizeSize and capacity of the list

StringSizeSize and capacity of key strings

7.13.23 TFPHashList.ForEachCall

Synopsis: Call a procedure for each element in the list

Declaration:
```
procedure ForEachCall(proc2call: TListCallback;arg: pointer)
procedure ForEachCall(proc2call: TListStaticCallback;arg: pointer)
```

Visibility: public

Description: `ForEachCall` loops over the items in the list and calls `proc2call`, passing it the item and `arg`.

7.13.24 TFPHashList.Capacity

Synopsis: Capacity of the list.

Declaration:
```
Property Capacity : Integer
```

Visibility: public

Access: Read,Write

Description: `Capacity` returns the current capacity of the list. The capacity is expanded as more elements are added to the list. If a good estimate of the number of elements that will be added to the list, the property can be set to a sufficiently large value to avoid reallocation of memory each time the list needs to grow.

See also: TFPHashList.Count (107), TFPHashList.Items (108)

7.13.25 TFPHashList.Count

Synopsis: Current number of elements in the list.

Declaration:
```
Property Count : Integer
```

Visibility: public

Access: Read,Write

Description: `Count` is the current number of elements in the list.

See also: TFPHashList.Capacity (107), TFPHashList.Items (108)

7.13.26 TFPHashList.Items

Synopsis: Indexed array with pointers

Declaration: `Property Items[Index: Integer]: Pointer; default`

Visibility: public

Access: Read,Write

Description: `Items` provides indexed access to the pointers, the index runs from 0 to Count-1 (107).

Errors: Specifying an invalid index will result in an exception.

See also: TFPHashList.Capacity (107), TFPHashList.Count (107)

7.13.27 TFPHashList.List

Synopsis: Low-level hash list

Declaration: `Property List : PHashItemList`

Visibility: public

Access: Read

Description: `List` exposes the low-level item list (84). It should not be used directly.

See also: TFPHashList.Strs (108), THashItemList (84)

7.13.28 TFPHashList.Strs

Synopsis: Low-level memory area with strings.

Declaration: `Property Strs : PChar`

Visibility: public

Access: Read

Description: `Strs` exposes the raw memory area with the strings.

See also: TFPHashList.List (108)

7.14 TFPHashObject

7.14.1 Description

`TFPHashObject` is a `TObject` descendent which is aware of the TFPHashObjectList (111) class. It has a name property and an owning list: if the name is changed, it will reposition itself in the list which owns it. It offers methods to change the owning list: the object will correctly remove itself from the list which currently owns it, and insert itself in the new list.

7.14.2 Method overview

Page	Property	Description
109	ChangeOwner	Change the list owning the object.
110	ChangeOwnerAndName	Simultaneously change the list owning the object and the name of the object.
109	Create	Create a named instance, and insert in a hash list.
109	CreateNotOwned	Create an instance not owned by any list.
110	Rename	Rename the object

7.14.3 Property overview

Page	Property	Access	Description
110	Hash	r	Hash value
110	Name	r	Current name of the object

7.14.4 TFPHashObject.CreateNotOwned

Synopsis: Create an instance not owned by any list.

Declaration:
```
constructor CreateNotOwned
```

Visibility: public

Description: `CreateNotOwned` creates an instance of `TFPHashObject` which is not owned by any TFPHashObjectList (111) hash list. It also has no name when created in this way.

See also: TFPHashObject.Name (110), TFPHashObject.ChangeOwner (109), TFPHashObject.ChangeOwnerAndName (110)

7.14.5 TFPHashObject.Create

Synopsis: Create a named instance, and insert in a hash list.

Declaration:
```
constructor Create(HashObjectList: TFPHashObjectList;
                   const s: shortstring)
```

Visibility: public

Description: `Create` creates an instance of `TFPHashObject`, gives it the name `S` and inserts it in the hash list HashObjectList (111).

See also: TFPHashObject.CreateNotOwned (109), TFPHashObject.ChangeOwner (109), TFPHashObject.Name (110)

7.14.6 TFPHashObject.ChangeOwner

Synopsis: Change the list owning the object.

Declaration:
```
procedure ChangeOwner(HashObjectList: TFPHashObjectList)
```

Visibility: public

Description: `ChangeOwner` can be used to move the object between hash lists: The object will be removed correctly from the hash list that currently owns it, and will be inserted in the list `HashObjectList`.

Errors: If an object with the same name already is present in the new hash list, an exception will be raised.

See also: TFPHashObject.ChangeOwnerAndName (110), TFPHashObject.Name (110)

7.14.7 TFPHashObject.ChangeOwnerAndName

Synopsis: Simultaneously change the list owning the object and the name of the object.

Declaration:
```
procedure ChangeOwnerAndName(HashObjectList: TFPHashObjectList;
                             const s: shortstring)
```

Visibility: public

Description: `ChangeOwnerAndName` can be used to move the object between hash lists: The object will be removed correctly from the hash list that currently owns it (using the current name), and will be inserted in the list `HashObjectList` with the new name `S`.

Errors: If the new name already is present in the new hash list, an exception will be raised.

See also: TFPHashObject.ChangeOwner (109), TFPHashObject.Name (110)

7.14.8 TFPHashObject.Rename

Synopsis: Rename the object

Declaration:
```
procedure Rename(const ANewName: shortstring)
```

Visibility: public

Description: `Rename` changes the name of the object, and notifies the hash list of this change.

Errors: If the new name already is present in the hash list, an exception will be raised.

See also: TFPHashObject.ChangeOwner (109), TFPHashObject.ChangeOwnerAndName (110), TFPHashObject.Name (110)

7.14.9 TFPHashObject.Name

Synopsis: Current name of the object

Declaration:
```
Property Name : shortstring
```

Visibility: public

Access: Read

Description: `Name` is the name of the object, it is stored in the hash list using this name as the key.

See also: TFPHashObject.Rename (110), TFPHashObject.ChangeOwnerAndName (110)

7.14.10 TFPHashObject.Hash

Synopsis: Hash value

Declaration:
```
Property Hash : LongWord
```

Visibility: public

Access: Read

Description: `Hash` is the hash value of the object in the hash list that owns it.

See also: TFPHashObject.Name (110)

7.15 TFPHashObjectList

7.15.1 Method overview

Page	Property	Description
112	Add	Add a new key/data pair to the list
112	Clear	Clear the list
111	Create	Create a new instance of the hashlist
113	Delete	Delete an object from the list.
111	Destroy	Removes an instance of the hashlist from the heap
113	Expand	Expand the list
114	Extract	Extract a object instance from the list
114	Find	Find data associated with key
115	FindIndexOf	Return index of named object.
115	FindInstanceOf	Search an instance of a certain class
115	FindWithHash	Find first element with given name and hash value
116	ForEachCall	Call a procedure for each object in the list
113	GetNextCollision	Get next collision number
113	HashOfIndex	Return the hash valye of an object by index
114	IndexOf	Return the index of the object instance
112	NameOfIndex	Returns the key name of an object by index
116	Pack	Remove nil object instances from the list
114	Remove	Remove first occurrence of a object instance
115	Rename	Rename a key
116	ShowStatistics	Return some statistics for the list.

7.15.2 Property overview

Page	Property	Access	Description
116	Capacity	rw	Capacity of the list.
117	Count	rw	Current number of elements in the list.
117	Items	rw	Indexed array with object instances
117	List	r	Low-level hash list
117	OwnsObjects	rw	Does the list own the objects it contains

7.15.3 TFPHashObjectList.Create

Synopsis: Create a new instance of the hashlist

Declaration: `constructor Create(FreeObjects: Boolean)`

Visibility: public

Description: `Create` creates a new instance of `TFPHashObjectList` on the heap and sets the hash capacity to 1.

If `FreeObjects` is `True` (the default), then the list owns the objects: when an object is removed from the list, it is destroyed (freed from memory). Clearing the list will free all objects in the list.

See also: TFPHashObjectList.Destroy (111), TFPHashObjectList.OwnsObjects (117)

7.15.4 TFPHashObjectList.Destroy

Synopsis: Removes an instance of the hashlist from the heap

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the memory structures maintained by the hashlist and removes the `TFPHashObjectList` instance from the heap. If the list owns its objects, they are freed from memory as well.

`Destroy` should not be called directly, it's better to use `Free` or `FreeAndNil` instead.

See also: TFPHashObjectList.Create (111), TFPHashObjectList.Clear (112)

7.15.5 TFPHashObjectList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` removes all objects from the list. It does not free the objects themselves, unless OwnsObjects (117) is `True`. It always frees all memory needed to contain the objects.

Errors: None.

See also: TFPHashObjectList.Extract (114), TFPHashObjectList.Remove (114), TFPHashObjectList.Delete (113), TFPHashObjectList.Add (112)

7.15.6 TFPHashObjectList.Add

Synopsis: Add a new key/data pair to the list

Declaration: `function Add(const AName: shortstring;AObject: TObject) : Integer`

Visibility: public

Description: `Add` adds a new object instance (`AObject`) with key `AName` to the list. It returns the position of the object in the list.

Errors: If not enough memory is available to hold the key and data, an exception may be raised. If an object with this name already exists in the list, an exception is raised.

See also: TFPHashObjectList.Extract (114), TFPHashObjectList.Remove (114), TFPHashObjectList.Delete (113)

7.15.7 TFPHashObjectList.NameOfIndex

Synopsis: Returns the key name of an object by index

Declaration: `function NameOfIndex(Index: Integer) : ShortString`

Visibility: public

Description: `NameOfIndex` returns the key name of the object at position `Index`.

Errors: If `Index` is out of the valid range, an exception is raised.

See also: TFPHashObjectList.HashOfIndex (113), TFPHashObjectList.Find (114), TFPHashObjectList.FindIndexOf (115), TFPHashObjectList.FindWithHash (115)

7.15.8 TFPHashObjectList.HashOfIndex

Synopsis: Return the hash valye of an object by index

Declaration: `function HashOfIndex(Index: Integer) : LongWord`

Visibility: public

Description: `HashOfIndex` returns the hash value of the object at position `Index`.

Errors: If `Index` is out of the valid range, an exception is raised.

See also: TFPHashObjectList.HashOfName (111), TFPHashObjectList.Find (114), TFPHashObjectList.FindIndexOf (115), TFPHashObjectList.FindWithHash (115)

7.15.9 TFPHashObjectList.GetNextCollision

Synopsis: Get next collision number

Declaration: `function GetNextCollision(Index: Integer) : Integer`

Visibility: public

Description: Get next collision number

Errors:

7.15.10 TFPHashObjectList.Delete

Synopsis: Delete an object from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` deletes the object at position `Index`. If OwnsObjects (117) is `True`, then the object itself is also freed from memory.

See also: TFPHashObjectList.Extract (114), TFPHashObjectList.Remove (114), TFPHashObjectList.Add (112), TFPHashObjectList.OwnsObjects (117)

7.15.11 TFPHashObjectList.Expand

Synopsis: Expand the list

Declaration: `function Expand : TFPHashObjectList`

Visibility: public

Description: `Expand` enlarges the capacity of the list if the maximum capacity was reached. It returns itself.

Errors: If not enough memory is available, an exception may be raised.

See also: TFPHashObjectList.Clear (112)

7.15.12 TFPHashObjectList.Extract

Synopsis: Extract a object instance from the list

Declaration: `function Extract(Item: TObject) : TObject`

Visibility: public

Description: `Extract` removes the data object from the list, if it is in the list. It returns the object instance if it was removed from the list, `Nil` otherwise. The object is *not* freed from memory, regardless of the value of OwnsObjects (117).

`Extract` does a linear search, and is not very efficient.

See also: TFPHashObjectList.Delete (113), TFPHashObjectList.Remove (114), TFPHashObjectList.Clear (112)

7.15.13 TFPHashObjectList.Remove

Synopsis: Remove first occurrence of a object instance

Declaration: `function Remove(AObject: TObject) : Integer`

Visibility: public

Description: `Remove` removes the first occurence of the object instance `Item` in the list, if it is present. The return value is the location of the removed object instance, or `-1` if no object instance was removed.

If OwnsObjects (117) is `True`, then the object itself is also freed from memory.

See also: TFPHashObjectList.Delete (113), TFPHashObjectList.Clear (112), TFPHashObjectList.Extract (114)

7.15.14 TFPHashObjectList.IndexOf

Synopsis: Return the index of the object instance

Declaration: `function IndexOf(AObject: TObject) : Integer`

Visibility: public

Description: `IndexOf` returns the index of the first occurrence of object instance `AObject`. If the object is not in the list, -1 is returned.

The performed search is linear, and not very efficient.

See also: TFPHashObjectList.HashOfIndex (113), TFPHashObjectList.NameOfIndex (112), TFPHashObjectList.Find (114), TFPHashObjectList.FindIndexOf (115), TFPHashObjectList.FindWithHash (115)

7.15.15 TFPHashObjectList.Find

Synopsis: Find data associated with key

Declaration: `function Find(const s: shortstring) : TObject`

Visibility: public

Description: `Find` searches (using the hash) for the data object associated with key `AName` and returns the data object instance associated with it. If the object is not found, `Nil` is returned. It uses the hash value of the key to perform the search.

See also: TFPHashObjectList.HashOfIndex (113), TFPHashObjectList.NameOfIndex (112), TFPHashObjectList.IndexOf (114), TFPHashObjectList.FindIndexOf (115), TFPHashObjectList.FindWithHash (115)

7.15.16 TFPHashObjectList.FindIndexOf

Synopsis: Return index of named object.

Declaration:
```
function FindIndexOf(const s: shortstring) : Integer
```

Visibility: public

Description: `FindIndexOf` returns the index of the key `AName`, or -1 if the key does not exist in the list. It uses the hash value to search for the key.

See also: TFPHashObjectList.HashOfIndex (113), TFPHashObjectList.NameOfIndex (112), TFPHashObjectList.IndexOf (114), TFPHashObjectList.Find (114), TFPHashObjectList.FindWithHash (115)

7.15.17 TFPHashObjectList.FindWithHash

Synopsis: Find first element with given name and hash value

Declaration:
```
function FindWithHash(const AName: shortstring;AHash: LongWord)
                            : Pointer
```

Visibility: public

Description: `FindWithHash` searches for the object with key `AName`. It uses the provided hash value `AHash` to perform the search. If the object exists, the data object instance is returned, if not, the result is `Nil`.

See also: TFPHashObjectList.HashOfIndex (113), TFPHashObjectList.NameOfIndex (112), TFPHashObjectList.IndexOf (114), TFPHashObjectList.Find (114), TFPHashObjectList.FindIndexOf (115)

7.15.18 TFPHashObjectList.Rename

Synopsis: Rename a key

Declaration:
```
function Rename(const AOldName: shortstring;const ANewName: shortstring)
                    : Integer
```

Visibility: public

Description: `Rename` renames key `AOldname` to `ANewName`. The hash value is recomputed and the object is moved in the list to it's new position.

Errors: If an object with `ANewName` already exists, an exception will be raised.

7.15.19 TFPHashObjectList.FindInstanceOf

Synopsis: Search an instance of a certain class

Declaration:
```
function FindInstanceOf(AClass: TClass;AExact: Boolean;
                            AStartAt: Integer) : Integer
```

Visibility: public

Description: `FindInstanceOf` searches the list for an instance of class `AClass`. It starts searching at position `AStartAt`. If `AExact` is `True`, only instances of class `AClass` are considered. If `AExact` is `False`, then descendent classes of `AClass` are also taken into account when searching. If no instance is found, `Nil` is returned.

7.15.20 TFPHashObjectList.Pack

Synopsis: Remove nil object instances from the list

Declaration: `procedure Pack`

Visibility: public

Description: `Pack` removes all `Nil` objects from the list, and frees all unused memory.

See also: TFPHashObjectList.Clear (112)

7.15.21 TFPHashObjectList.ShowStatistics

Synopsis: Return some statistics for the list.

Declaration: `procedure ShowStatistics`

Visibility: public

Description: `ShowStatistics` prints some information about the hash list to standard output. It prints the following values:

HashSizeSize of the hash table

HashMeanMean hash value

HashStdDevStandard deviation of hash values

ListSizeSize and capacity of the list

StringSizeSize and capacity of key strings

7.15.22 TFPHashObjectList.ForEachCall

Synopsis: Call a procedure for each object in the list

Declaration:
```
procedure ForEachCall(proc2call: TObjectListCallback;arg: pointer)
procedure ForEachCall(proc2call: TObjectListStaticCallback;arg: pointer)
```

Visibility: public

Description: `ForEachCall` loops over the objects in the list and calls `proc2call`, passing it the object and `arg`.

7.15.23 TFPHashObjectList.Capacity

Synopsis: Capacity of the list.

Declaration: `Property Capacity : Integer`

Visibility: public

Access: Read,Write

Description: `Capacity` returns the current capacity of the list. The capacity is expanded as more elements are added to the list. If a good estimate of the number of elements that will be added to the list, the property can be set to a sufficiently large value to avoid reallocation of memory each time the list needs to grow.

See also: TFPHashObjectList.Count (117), TFPHashObjectList.Items (117)

7.15.24 TFPHashObjectList.Count

Synopsis: Current number of elements in the list.

Declaration: `Property Count : Integer`

Visibility: public

Access: Read,Write

Description: `Count` is the current number of elements in the list.

See also: TFPHashObjectList.Capacity (116), TFPHashObjectList.Items (117)

7.15.25 TFPHashObjectList.OwnsObjects

Synopsis: Does the list own the objects it contains

Declaration: `Property OwnsObjects : Boolean`

Visibility: public

Access: Read,Write

Description: `OwnsObjects` determines what to do when an object is removed from the list: if it is `True` (the default), then the list owns the objects: when an object is removed from the list, it is destroyed (freed from memory). Clearing the list will free all objects in the list.

The value of `OwnsObjects` is set when the hash list is created, and cannot be changed during the lifetime of the hash list.

See also: TFPHashObjectList.Create (111)

7.15.26 TFPHashObjectList.Items

Synopsis: Indexed array with object instances

Declaration: `Property Items[Index: Integer]: TObject; default`

Visibility: public

Access: Read,Write

Description: `Items` provides indexed access to the object instances, the index runs from 0 to Count-1 (117).

Errors: Specifying an invalid index will result in an exception.

See also: TFPHashObjectList.Capacity (116), TFPHashObjectList.Count (117)

7.15.27 TFPHashObjectList.List

Synopsis: Low-level hash list

Declaration: `Property List : TFPHashList`

Visibility: public

Access: Read

Description: `List` exposes the low-level hash list (101). It should not be used directly.

See also: TFPHashList (101)

7.16 TFPObjectHashTable

7.16.1 Description

`TFPStringHashTable` is a TFPCustomHashTable (94) descendent which stores object instances together with the keys. In case the data associated with the keys are strings themselves, it's better to use TFPStringHashTable (127), or for arbitrary pointer data, TFPDataHashTable (100) is more suitable. The objects are exposed with their keys through the Items (119) property.

7.16.2 Method overview

Page	Property	Description
119	Add	Add a new object to the hash table
118	Create	Create a new instance of `TFPObjectHashTable`
118	CreateWith	Create a new hash table with given size and hash function

7.16.3 Property overview

Page	Property	Access	Description
119	Items	rw	Key-based access to the objects
119	OwnsObjects	rw	Does the hash table own the objects ?

7.16.4 TFPObjectHashTable.Create

Synopsis: Create a new instance of `TFPObjectHashTable`

Declaration:
```
constructor Create(AOwnsObjects: Boolean)
```

Visibility: public

Description: `Create` creates a new instance of `TFPObjectHashTable` on the heap. It sets the OwnsObjects (119) property to `AOwnsObjects`, and then calls the inherited `Create`. If `AOwnsObjects` is set to `True`, then the hash table owns the objects: whenever an object is removed from the list, it is automatically freed.

Errors: If not enough memory is available on the heap, an exception may be raised.

See also: TFPObjectHashTable.OwnsObjects (119), TFPObjectHashTable.CreateWith (118), TFPObjectHashTable.Items (119)

7.16.5 TFPObjectHashTable.CreateWith

Synopsis: Create a new hash table with given size and hash function

Declaration:
```
constructor CreateWith(AHashTableSize: LongWord;
                       aHashFunc: THashFunction;AOwnsObjects: Boolean)
```

Visibility: public

Description: `CreateWith` sets the OwnsObjects (119) property to `AOwnsObjects`, and then calls the inherited `CreateWith`. If `AOwnsObjects` is set to `True`, then the hash table owns the objects: whenever an object is removed from the list, it is automatically freed.

This constructor should be used when a table size and hash algorithm should be specified that differ from the default table size and hash algorithm.

Errors: If not enough memory is available on the heap, an exception may be raised.

See also: TFPObjectHashTable.OwnsObjects (119), TFPObjectHashTable.Create (118), TFPObjectHashTable.Items (119)

7.16.6 TFPObjectHashTable.Add

Synopsis: Add a new object to the hash table

Declaration: `procedure Add(const aKey: String;AItem: TObject);  Virtual`

Visibility: public

Description: `Add` adds the object `AItem` to the hash table, and associates it with key `aKey`.

Errors: If the key `aKey` is already in the hash table, an exception will be raised.

See also: TFPObjectHashTable.Items (119)

7.16.7 TFPObjectHashTable.Items

Synopsis: Key-based access to the objects

Declaration: `Property Items[index: String]: TObject; default`

Visibility: public

Access: Read,Write

Description: `Items` provides access to the objects in the hash table using their key: the array index `Index` is the key. A key which is not present will result in an `Nil` instance.

See also: TFPObjectHashTable.Add (119)

7.16.8 TFPObjectHashTable.OwnsObjects

Synopsis: Does the hash table own the objects ?

Declaration: `Property OwnsObjects : Boolean`

Visibility: public

Access: Read,Write

Description: `OwnsObjects` determines what happens with objects which are removed from the hash table: if `True`, then removing an object from the hash list will free the object. If `False`, the object is not freed. Note that way in which the object is removed is not relevant: be it `Delete`, `Remove` or `Clear`.

See also: TFPObjectHashTable.Create (118), TFPObjectHashTable.Items (119)

7.17 TFPObjectList

7.17.1 Description

`TFPObjectList` is a TFPList (**??**) based list which has as the default array property TObjects (**??**) instead of pointers. By default it also manages the objects: when an object is deleted or removed from the list, it is automatically freed. This behaviour can be disabled when the list is created.

In difference with TObjectList (132), `TFPObjectList` offers no notification mechanism of list operations, allowing it to be faster than `TObjectList`. For the same reason, it is also not a descendent of `TFPList` (although it uses one internally).

7.17.2 Method overview

Page	Property	Description
121	Add	Add an object to the list.
125	Assign	Copy the contents of a list.
121	Clear	Clear all elements in the list.
120	Create	Create a new object list
121	Delete	Delete an element from the list.
121	Destroy	Clears the list and destroys the list instance
122	Exchange	Exchange the location of two objects
122	Expand	Expand the capacity of the list.
122	Extract	Extract an object from the list
123	FindInstanceOf	Search for an instance of a certain class
124	First	Return the first non-nil object in the list
125	ForEachCall	For each object in the list, call a method or procedure, passing it the object.
123	IndexOf	Search for an object in the list
123	Insert	Insert a new object in the list
124	Last	Return the last non-nil object in the list.
124	Move	Move an object to another location in the list.
125	Pack	Remove all `Nil` references from the list
123	Remove	Remove an item from the list.
125	Sort	Sort the list of objects

7.17.3 Property overview

Page	Property	Access	Description
126	Capacity	rw	Capacity of the list
126	Count	rw	Number of elements in the list.
127	Items	rw	Indexed access to the elements of the list.
127	List	r	Internal list used to keep the objects.
126	OwnsObjects	rw	Should the list free elements when they are removed.

7.17.4 TFPObjectList.Create

Synopsis: Create a new object list

Declaration:
```
constructor Create
constructor Create(FreeObjects: Boolean)
```

Visibility: public

Description: `Create` instantiates a new object list. The `FreeObjects` parameter determines whether objects that are removed from the list should also be freed from memory. By default this is `True`. This behaviour can be changed after the list was instantiated.

Errors: None.

See also: TFPObjectList.Destroy (121), TFPObjectList.OwnsObjects (126), TObjectList (132)

7.17.5 TFPObjectList.Destroy

Synopsis: Clears the list and destroys the list instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` clears the list, freeing all objects in the list if OwnsObjects (126) is `True`.

See also: TFPObjectList.OwnsObjects (126), TObjectList.Create (133)

7.17.6 TFPObjectList.Clear

Synopsis: Clear all elements in the list.

Declaration: `procedure Clear`

Visibility: public

Description: Removes all objects from the list, freeing all objects in the list if OwnsObjects (126) is `True`.

See also: TObjectList.Destroy (132)

7.17.7 TFPObjectList.Add

Synopsis: Add an object to the list.

Declaration: `function Add(AObject: TObject) : Integer`

Visibility: public

Description: `Add` adds `AObject` to the list and returns the index of the object in the list.

Note that when OwnsObjects (126) is `True`, an object should not be added twice to the list: this will result in memory corruption when the object is freed (as it will be freed twice). The `Add` method does not check this, however.

Errors: None.

See also: TFPObjectList.OwnsObjects (126), TFPObjectList.Delete (121)

7.17.8 TFPObjectList.Delete

Synopsis: Delete an element from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` removes the object at index `Index` from the list. When OwnsObjects (126) is `True`, the object is also freed.

Errors: An access violation may occur when OwnsObjects (126) is `True` and either the object was freed externally, or when the same object is in the same list twice.

See also: TTFPObjectList.Remove (81), TFPObjectList.Extract (122), TFPObjectList.OwnsObjects (126), TTFPObjectList.Add (81), TTFPObjectList.Clear (81)

7.17.9 TFPObjectList.Exchange

Synopsis: Exchange the location of two objects

Declaration: `procedure Exchange(Index1: Integer;Index2: Integer)`

Visibility: public

Description: `Exchange` exchanges the objects at indexes `Index1` and `Index2` in a direct operation (i.e. no delete/add is performed).

Errors: If either `Index1` or `Index2` is invalid, an exception will be raised.

See also: TTFPObjectList.Add (81), TTFPObjectList.Delete (81)

7.17.10 TFPObjectList.Expand

Synopsis: Expand the capacity of the list.

Declaration: `function Expand : TFPObjectList`

Visibility: public

Description: `Expand` increases the capacity of the list. It calls #rtl.classes.tfplist.expand (**??**) and then returns a reference to itself.

Errors: If there is not enough memory to expand the list, an exception will be raised.

See also: TFPObjectList.Pack (125), TFPObjectList.Clear (121), #rtl.classes.tfplist.expand (**??**)

7.17.11 TFPObjectList.Extract

Synopsis: Extract an object from the list

Declaration: `function Extract(Item: TObject) : TObject`

Visibility: public

Description: `Extract` removes `Item` from the list, if it is present in the list. It returns `Item` if it was found, `Nil` if item was not present in the list.

Note that the object is not freed, and that only the first found object is removed from the list.

Errors: None.

See also: TFPObjectList.Pack (125), TFPObjectList.Clear (121), TFPObjectList.Remove (123), TFPObjectList.Delete (121)

7.17.12 TFPObjectList.Remove

Synopsis: Remove an item from the list.

Declaration:
```
function Remove(AObject: TObject) : Integer
```

Visibility: public

Description: `Remove` removes `Item` from the list, if it is present in the list. It frees `Item` if OwnsObjects (126) is `True`, and returns the index of the object that was found in the list, or -1 if the object was not found.

Note that only the first found object is removed from the list.

Errors: None.

See also: TFPObjectList.Pack (125), TFPObjectList.Clear (121), TFPObjectList.Delete (121), TFPObjectList.Extract (122)

7.17.13 TFPObjectList.IndexOf

Synopsis: Search for an object in the list

Declaration:
```
function IndexOf(AObject: TObject) : Integer
```

Visibility: public

Description: `IndexOf` searches for the presence of `AObject` in the list, and returns the location (index) in the list. The index is 0-based, and -1 is returned if `AObject` was not found in the list.

Errors: None.

See also: TFPObjectList.Items (127), TFPObjectList.Remove (123), TFPObjectList.Extract (122)

7.17.14 TFPObjectList.FindInstanceOf

Synopsis: Search for an instance of a certain class

Declaration:
```
function FindInstanceOf(AClass: TClass;AExact: Boolean;
                        AStartAt: Integer) : Integer
```

Visibility: public

Description: `FindInstanceOf` will look through the instances in the list and will return the first instance which is a descendent of class `AClass` if `AExact` is `False`. If `AExact` is true, then the instance should be of class `AClass`.

If no instance of the requested class is found, `Nil` is returned.

Errors: None.

See also: TFPObjectList.IndexOf (123)

7.17.15 TFPObjectList.Insert

Synopsis: Insert a new object in the list

Declaration: `procedure Insert(Index: Integer;AObject: TObject)`

Visibility: public

Description: `Insert` inserts `AObject` at position `Index` in the list. All elements in the list after this position are shifted. The index is zero based, i.e. an insert at position 0 will insert an object at the first position of the list.

Errors: None.

See also: TFPObjectList.Add (121), TFPObjectList.Delete (121)

7.17.16 TFPObjectList.First

Synopsis: Return the first non-nil object in the list

Declaration: `function First : TObject`

Visibility: public

Description: `First` returns a reference to the first non-`Nil` element in the list. If no non-`Nil` element is found, `Nil` is returned.

Errors: None.

See also: TFPObjectList.Last (124), TFPObjectList.Pack (125)

7.17.17 TFPObjectList.Last

Synopsis: Return the last non-nil object in the list.

Declaration: `function Last : TObject`

Visibility: public

Description: `Last` returns a reference to the last non-`Nil` element in the list. If no non-`Nil` element is found, `Nil` is returned.

Errors: None.

See also: TFPObjectList.First (124), TFPObjectList.Pack (125)

7.17.18 TFPObjectList.Move

Synopsis: Move an object to another location in the list.

Declaration: `procedure Move(CurIndex: Integer;NewIndex: Integer)`

Visibility: public

Description: `Move` moves the object at current location `CurIndex` to location `NewIndex`. Note that the `NewIndex` is determined *after* the object was removed from location `CurIndex`, and can hence be shifted with 1 position if `CurIndex` is less than `NewIndex`.

Contrary to exchange (122), the move operation is done by extracting the object from it's current location and inserting it at the new location.

Errors: If either `CurIndex` or `NewIndex` is out of range, an exception may occur.

See also: TFPObjectList.Exchange (122), TFPObjectList.Delete (121), TFPObjectList.Insert (123)

7.17.19 TFPObjectList.Assign

Synopsis: Copy the contents of a list.

Declaration: `procedure Assign(Obj: TFPObjectList)`

Visibility: public

Description: `Assign` copies the contents of `Obj` if `Obj` is of type `TFPObjectList`

Errors: None.

7.17.20 TFPObjectList.Pack

Synopsis: Remove all `Nil` references from the list

Declaration: `procedure Pack`

Visibility: public

Description: `Pack` removes all `Nil` elements from the list.

Errors: None.

See also: TFPObjectList.First (124), TFPObjectList.Last (124)

7.17.21 TFPObjectList.Sort

Synopsis: Sort the list of objects

Declaration: `procedure Sort(Compare: TListSortCompare)`

Visibility: public

Description: `Sort` will perform a quick-sort on the list, using `Compare` as the compare algorithm. This function should accept 2 pointers and should return the following result:

less than 0 If the first pointer comes before the second.

equal to 0 If the pointers have the same value.

larger than 0 If the first pointer comes after the second.

The function should be able to deal with `Nil` values.

Errors: None.

See also: #rtl.classes.TList.Sort (**??**)

7.17.22 TFPObjectList.ForEachCall

Synopsis: For each object in the list, call a method or procedure, passing it the object.

Declaration:
```
procedure ForEachCall(proc2call: TObjectListCallback;arg: pointer)
procedure ForEachCall(proc2call: TObjectListStaticCallback;arg: pointer)
```

Visibility: public

Description: `ForEachCall` loops through all objects in the list, and calls `proc2call`, passing it the object in the list. Additionally, `arg` is also passed to the procedure. `Proc2call` can be a plain procedure or can be a method of a class.

Errors: None.

See also: TObjectListStaticCallback (85), TObjectListCallback (84)

7.17.23 TFPObjectList.Capacity

Synopsis: Capacity of the list

Declaration:
```
Property Capacity : Integer
```

Visibility: public

Access: Read,Write

Description: `Capacity` is the number of elements that the list can contain before it needs to expand itself, i.e., reserve more memory for pointers. It is always equal or larger than Count (126).

See also: TFPObjectList.Count (126)

7.17.24 TFPObjectList.Count

Synopsis: Number of elements in the list.

Declaration:
```
Property Count : Integer
```

Visibility: public

Access: Read,Write

Description: `Count` is the number of elements in the list. Note that this includes `Nil` elements.

See also: TFPObjectList.Capacity (126)

7.17.25 TFPObjectList.OwnsObjects

Synopsis: Should the list free elements when they are removed.

Declaration:
```
Property OwnsObjects : Boolean
```

Visibility: public

Access: Read,Write

Description: `OwnsObjects` determines whether the objects in the list should be freed when they are removed (not extracted) from the list, or when the list is cleared. If the property is `True` then they are freed. If the property is `False` the elements are not freed.

The value is usually set in the constructor, and is seldom changed during the lifetime of the list. It defaults to `True`.

See also: TFPObjectList.Create (120), TFPObjectList.Delete (121), TFPObjectList.Remove (123), TFPObjectList.Clear (121)

7.17.26 TFPObjectList.Items

Synopsis: Indexed access to the elements of the list.

Declaration: `Property Items[Index: Integer]: TObject; default`

Visibility: public

Access: Read,Write

Description: `Items` is the default property of the list. It provides indexed access to the elements in the list. The index `Index` is zero based, i.e., runs from 0 (zero) to `Count-1`.

See also: TFPObjectList.Count (126)

7.17.27 TFPObjectList.List

Synopsis: Internal list used to keep the objects.

Declaration: `Property List : TFPList`

Visibility: public

Access: Read

Description: `List` is a reference to the TFPList (**??**) instance used to manage the elements in the list.

See also: #rtl.classes.tfplist (**??**)

7.18 TFPStringHashTable

7.18.1 Description

`TFPStringHashTable` is a TFPCustomHashTable (94) descendent which stores simple strings together with the keys. In case the data associated with the keys are objects, it's better to use TFPObjectHashTable (118), or for arbitrary pointer data, TFPDataHashTable (100) is more suitable. The strings are exposed with their keys through the Items (128) property.

7.18.2 Method overview

Page	Property	Description
127	Add	Add a new string to the hash list

7.18.3 Property overview

Page	Property	Access	Description
128	Items	rw	Key based access to the strings in the hash table

7.18.4 TFPStringHashTable.Add

Synopsis: Add a new string to the hash list

Declaration: `procedure Add(const aKey: String;const aItem: String);  Virtual`

Visibility: public

Description: `Add` adds a new string `AItem` to the hash list with key `AKey`.

Errors: If a string with key `Akey` already exists in the hash table, an exception will be raised.

See also: TFPStringHashTable.Items (128)

7.18.5 TFPStringHashTable.Items

Synopsis: Key based access to the strings in the hash table

Declaration: `Property Items[index: String]: String; default`

Visibility: public

Access: Read,Write

Description: `Items` provides access to the strings in the hash table using their key: the array index `Index` is the key. A key which is not present will result in an empty string.

See also: TFPStringHashTable.Add (127)

7.19 THTCustomNode

7.19.1 Description

`THTCustomNode` is used by the TFPCustomHashTable (94) class to store the keys and associated values.

7.19.2 Method overview

Page	Property	Description
128	CreateWith	Create a new instance of `THTCustomNode`
129	HasKey	Check whether this node matches the given key.

7.19.3 Property overview

Page	Property	Access	Description
129	Key	r	Key value associated with this hash item.

7.19.4 THTCustomNode.CreateWith

Synopsis: Create a new instance of `THTCustomNode`

Declaration: `constructor CreateWith(const AString: String)`

Visibility: public

Description: `CreateWith` creates a new instance of `THTCustomNode` and stores the string `AString` in it. It should never be necessary to call this method directly, it will be called by the TFPHashTable (81) class when needed.

Errors: If no more memory is available, an exception may be raised.

See also: TFPHashTable (81)

7.19.5 THTCustomNode.HasKey

Synopsis: Check whether this node matches the given key.

Declaration: `function HasKey(const AKey: String) : Boolean`

Visibility: public

Description: `HasKey` checks whether this node matches the given key `AKey`, by comparing it with the stored key. It returns `True` if it does, `False` if not.

Errors: None.

See also: THTCustomNode.Key (129)

7.19.6 THTCustomNode.Key

Synopsis: Key value associated with this hash item.

Declaration: `Property Key : String`

Visibility: public

Access: Read

Description: `Key` is the key value associated with this hash item. It is stored when the item is created, and is read-only.

See also: THTCustomNode.CreateWith (128)

7.20 THTDataNode

7.20.1 Description

`THTDataNode` is used by TDataHashTable (81) to store the hash items in. It simply holds the data pointer.

It should not be necessary to use `THTDataNode` directly, it's only for inner use by `TFPDataHashTable`

7.20.2 Property overview

Page	Property	Access	Description
129	Data	rw	Data pointer

7.20.3 THTDataNode.Data

Synopsis: Data pointer

Declaration: `Property Data : pointer`

Visibility: public

Access: Read,Write

Description: Pointer containing the user data associated with the hash value.

7.21 THTObjectNode

7.21.1 Description

`THTObjectNode` is a THTCustomNode (128) descendent which holds the data in the TFPObjectHashTable (118) hash table. It exposes a data string.

It should not be necessary to use `THTObjectNode` directly, it's only for inner use by `TFPObjectHashTable`

7.21.2 Property overview

Page	Property	Access	Description
130	Data	rw	Object instance

7.21.3 THTObjectNode.Data

Synopsis: Object instance

Declaration: `Property Data : TObject`

Visibility: public

Access: Read,Write

Description: `Data` is the object instance associated with the key value. It is exposed in TFPObjectHashTable.Items (119)

See also: TFPObjectHashTable (118), TFPObjectHashTable.Items (119), THTOwnedObjectNode (130)

7.22 THTOwnedObjectNode

7.22.1 Description

`THTOwnedObjectNode` is used instead of THTObjectNode (130) in case TFPObjectHashTable (118) owns it's objects. When this object is destroyed, the associated data object is also destroyed.

7.22.2 Method overview

Page	Property	Description
130	Destroy	Destroys the node and the object.

7.22.3 THTOwnedObjectNode.Destroy

Synopsis: Destroys the node and the object.

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` first frees the data object, and then only frees itself.

See also: THTOwnedObjectNode (130), TFPObjectHashTable.OwnsObjects (119)

7.23 THTStringNode

7.23.1 Description

`THTStringNode` is a THTCustomNode (128) descendent which holds the data in the TFPString-HashTable (127) hash table. It exposes a data string.

It should not be necessary to use `THTStringNode` directly, it's only for inner use by `TFPStringHashTable`

7.23.2 Property overview

Page	Property	Access	Description
131	Data	rw	String data

7.23.3 THTStringNode.Data

Synopsis: String data

Declaration: `Property Data : String`

Visibility: public

Access: Read,Write

Description: `Data` is the data of this has node. The data is a string, associated with the key. It is also exposed in TFPStringHashTable.Items (128)

See also: TFPStringHashTable (127)

7.24 TObjectBucketList

7.24.1 Description

`TObjectBucketList` is a class that redefines the associative `Data` array using `TObject` instead of `Pointer`. It also adds some overloaded versions of the `Add` and `Remove` calls using `TObject` instead of `Pointer` for the argument and result types.

7.24.2 Method overview

Page	Property	Description
131	Add	Add an object to the list
132	Remove	Remove an object from the list

7.24.3 Property overview

Page	Property	Access	Description
132	Data	rw	Associative array of data items

7.24.4 TObjectBucketList.Add

Synopsis: Add an object to the list

Declaration: `function Add(AItem: TObject;AData: TObject) : TObject`

Visibility: public

Description: `Add` adds `AItem` to the list and associated `AData` with it.

See also: TObjectBucketList.Data (132), TObjectBucketList.Remove (132)

7.24.5 TObjectBucketList.Remove

Synopsis: Remove an object from the list

Declaration: `function Remove(AItem: TObject) : TObject`

Visibility: public

Description: `Remove` removes the object `AItem` from the list. It returns the `Data` object which was associated with the item. If `AItem` was not in the list, then `Nil` is returned.

See also: TObjectBucketList.Add (131), TObjectBucketList.Data (132)

7.24.6 TObjectBucketList.Data

Synopsis: Associative array of data items

Declaration: `Property Data[AItem: TObject]: TObject; default`

Visibility: public

Access: Read,Write

Description: `Data` provides associative access to the data in the list: it returns the data object associated with the `AItem` object. If the `AItem` object is not in the list, an `EListError` exception is raised.

See also: TObjectBucketList.Add (131)

7.25 TObjectList

7.25.1 Description

`TObjectList` is a TList (**??**) descendent which has as the default array property TObjects (**??**) instead of pointers. By default it also manages the objects: when an object is deleted or removed from the list, it is automatically freed. This behaviour can be disabled when the list is created.

In difference with TFPObjectList (120), `TObjectList` offers a notification mechanism of list change operations: insert, delete. This slows down bulk operations, so if the notifications are not needed, `TObjectList` may be more appropriate.

7.25.2 Method overview

Page	Property	Description
133	Add	Add an object to the list.
133	create	Create a new object list.
133	Extract	Extract an object from the list.
134	FindInstanceOf	Search for an instance of a certain class
135	First	Return the first non-nil object in the list
134	IndexOf	Search for an object in the list
135	Insert	Insert an object in the list.
135	Last	Return the last non-nil object in the list.
134	Remove	Remove (and possibly free) an element from the list.

7.25.3 Property overview

Page	Property	Access	Description
136	Items	rw	Indexed access to the elements of the list.
135	OwnsObjects	rw	Should the list free elements when they are removed.

7.25.4 TObjectList.create

Synopsis: Create a new object list.

Declaration:
```
constructor create
constructor create(freeobjects: Boolean)
```

Visibility: public

Description: `Create` instantiates a new object list. The `FreeObjects` parameter determines whether objects that are removed from the list should also be freed from memory. By default this is `True`. This behaviour can be changed after the list was instantiated.

Errors: None.

See also: TObjectList.Destroy (132), TObjectList.OwnsObjects (135), TFPObjectList (120)

7.25.5 TObjectList.Add

Synopsis: Add an object to the list.

Declaration:
```
function Add(AObject: TObject) : Integer
```

Visibility: public

Description: `Add` overrides the TList (**??**) implementation to accept objects (`AObject`) instead of pointers.

The function returns the index of the position where the object was added.

Errors: If the list must be expanded, and not enough memory is available, an exception may be raised.

See also: TObjectList.Insert (135), #rtl.classes.TList.Delete (**??**), TObjectList.Extract (133), TObjectList.Remove (134)

7.25.6 TObjectList.Extract

Synopsis: Extract an object from the list.

Declaration:
```
function Extract(Item: TObject) : TObject
```

Visibility: public

Description: `Extract` removes the object `Item` from the list if it is present in the list. Contrary to Remove (134), `Extract` does not free the extracted element if OwnsObjects (135) is `True`

The function returns a reference to the item which was removed from the list, or `Nil` if no element was removed.

Errors: None.

See also: TObjectList.Remove (134)

7.25.7 TObjectList.Remove

Synopsis: Remove (and possibly free) an element from the list.

Declaration:
```
function Remove(AObject: TObject) : Integer
```

Visibility: public

Description: `Remove` removes `Item` from the list, if it is present in the list. It frees `Item` if OwnsObjects (135) is `True`, and returns the index of the object that was found in the list, or -1 if the object was not found.

Note that only the first found object is removed from the list.

Errors: None.

See also: TObjectList.Extract (133)

7.25.8 TObjectList.IndexOf

Synopsis: Search for an object in the list

Declaration:
```
function IndexOf(AObject: TObject) : Integer
```

Visibility: public

Description: `IndexOf` overrides the TList (**??**) implementation to accept an object instance instead of a pointer.

The function returns the index of the first match for `AObject` in the list, or -1 if no match was found.

Errors: None.

See also: TObjectList.FindInstanceOf (134)

7.25.9 TObjectList.FindInstanceOf

Synopsis: Search for an instance of a certain class

Declaration:
```
function FindInstanceOf(AClass: TClass;AExact: Boolean;
                        AStartAt: Integer) : Integer
```

Visibility: public

Description: `FindInstanceOf` will look through the instances in the list and will return the first instance which is a descendent of class `AClass` if `AExact` is `False`. If `AExact` is true, then the instance should be of class `AClass`.

If no instance of the requested class is found, `Nil` is returned.

Errors: None.

See also: TObjectList.IndexOf (134)

7.25.10 TObjectList.Insert

Synopsis: Insert an object in the list.

Declaration: `procedure Insert(Index: Integer;AObject: TObject)`

Visibility: public

Description: `Insert` inserts `AObject` in the list at position `Index`. The index is zero-based. This method overrides the implementation in TList (**??**) to accept objects instead of pointers.

Errors: If an invalid `Index` is specified, an exception is raised.

See also: TObjectList.Add (133), TObjectList.Remove (134)

7.25.11 TObjectList.First

Synopsis: Return the first non-nil object in the list

Declaration: `function First : TObject`

Visibility: public

Description: `First` returns a reference to the first non-`Nil` element in the list. If no non-`Nil` element is found, `Nil` is returned.

Errors: None.

See also: TObjectList.Last (135), TObjectList.Pack (132)

7.25.12 TObjectList.Last

Synopsis: Return the last non-nil object in the list.

Declaration: `function Last : TObject`

Visibility: public

Description: `Last` returns a reference to the last non-`Nil` element in the list. If no non-`Nil` element is found, `Nil` is returned.

Errors: None.

See also: TObjectList.First (135), TObjectList.Pack (132)

7.25.13 TObjectList.OwnsObjects

Synopsis: Should the list free elements when they are removed.

Declaration: `Property OwnsObjects : Boolean`

Visibility: public

Access: Read,Write

Description: `OwnsObjects` determines whether the objects in the list should be freed when they are removed (not extracted) from the list, or when the list is cleared. If the property is `True` then they are freed. If the property is `False` the elements are not freed.

The value is usually set in the constructor, and is seldom changed during the lifetime of the list. It defaults to `True`.

See also: TObjectList.Create (133), TObjectList.Delete (132), TObjectList.Remove (134), TObjectList.Clear (132)

7.25.14 TObjectList.Items

Synopsis: Indexed access to the elements of the list.

Declaration: `Property Items[Index: Integer]: TObject; default`

Visibility: public

Access: Read,Write

Description: `Items` is the default property of the list. It provides indexed access to the elements in the list. The index `Index` is zero based, i.e., runs from 0 (zero) to `Count-1`.

See also: #rtl.classes.TList.Count (**??**)

7.26 TObjectQueue

7.26.1 Method overview

Page	Property	Description
137	Peek	Look at the first object in the queue.
136	Pop	Pop the first element off the queue
136	Push	Push an object on the queue

7.26.2 TObjectQueue.Push

Synopsis: Push an object on the queue

Declaration: `function Push(AObject: TObject) : TObject`

Visibility: public

Description: `Push` pushes another object on the queue. It overrides the `Push` method as implemented in `TQueue` so it accepts only objects as arguments.

Errors: If not enough memory is available to expand the queue, an exception may be raised.

See also: TObjectQueue.Pop (136), TObjectQueue.Peek (137)

7.26.3 TObjectQueue.Pop

Synopsis: Pop the first element off the queue

Declaration: `function Pop : TObject`

Visibility: public

Description: `Pop` removes the first element in the queue, and returns a reference to the instance. If the queue is empty, `Nil` is returned.

Errors: None.

See also: TObjectQueue.Push (136), TObjectQueue.Peek (137)

7.26.4 TObjectQueue.Peek

Synopsis: Look at the first object in the queue.

Declaration: `function Peek : TObject`

Visibility: public

Description: `Peek` returns the first object in the queue, without removing it from the queue. If there are no more objects in the queue, `Nil` is returned.

Errors: None

See also: TObjectQueue.Push (136), TObjectQueue.Pop (136)

7.27 TObjectStack

7.27.1 Description

`TObjectStack` is a stack implementation which manages pointers only.

`TObjectStack` introduces no new behaviour, it simply overrides some methods to accept and/or return `TObject` instances instead of pointers.

7.27.2 Method overview

Page	Property	Description
138	Peek	Look at the top object in the stack.
137	Pop	Pop the top object of the stack.
137	Push	Push an object on the stack.

7.27.3 TObjectStack.Push

Synopsis: Push an object on the stack.

Declaration: `function Push(AObject: TObject) : TObject`

Visibility: public

Description: `Push` pushes another object on the stack. It overrides the `Push` method as implemented in `TStack` so it accepts only objects as arguments.

Errors: If not enough memory is available to expand the stack, an exception may be raised.

See also: TObjectStack.Pop (137), TObjectStack.Peek (138)

7.27.4 TObjectStack.Pop

Synopsis: Pop the top object of the stack.

Declaration: `function Pop : TObject`

Visibility: public

Description: `Pop` pops the top object of the stack, and returns the object instance. If there are no more objects on the stack, `Nil` is returned.

Errors: None

See also: TObjectStack.Push (137), TObjectStack.Peek (138)

7.27.5 TObjectStack.Peek

Synopsis: Look at the top object in the stack.

Declaration: `function Peek : TObject`

Visibility: public

Description: `Peek` returns the top object of the stack, without removing it from the stack. If there are no more objects on the stack, `Nil` is returned.

Errors: None

See also: TObjectStack.Push (137), TObjectStack.Pop (137)

7.28 TOrderedList

7.28.1 Description

`TOrderedList` provides the base class for TQueue (140) and TStack (140). It provides an interface for pushing and popping elements on or off the list, and manages the internal list of pointers.

Note that `TOrderedList` does not manage objects on the stack, i.e. objects are not freed when the ordered list is destroyed.

7.28.2 Method overview

Page	Property	Description
139	AtLeast	Check whether the list contains a certain number of elements.
139	Count	Number of elements on the list.
138	Create	Create a new ordered list
138	Destroy	Free an ordered list
140	Peek	Return the next element to be popped from the list.
140	Pop	Remove an element from the list.
139	Push	Push another element on the list.

7.28.3 TOrderedList.Create

Synopsis: Create a new ordered list

Declaration: `constructor Create`

Visibility: public

Description: `Create` instantiates a new ordered list. It initializes the internal pointer list.

Errors: None.

See also: TOrderedList.Destroy (138)

7.28.4 TOrderedList.Destroy

Synopsis: Free an ordered list

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the internal pointer list, and removes the `TOrderedList` instance from memory.

Errors: None.

See also: TOrderedList.Create (138)

7.28.5 TOrderedList.Count

Synopsis: Number of elements on the list.

Declaration: `function Count : Integer`

Visibility: public

Description: `Count` is the number of pointers in the list.

Errors: None.

See also: TOrderedList.AtLeast (139)

7.28.6 TOrderedList.AtLeast

Synopsis: Check whether the list contains a certain number of elements.

Declaration: `function AtLeast(ACount: Integer) : Boolean`

Visibility: public

Description: `AtLeast` returns `True` if the number of elements in the list is equal to or bigger than `ACount`. It returns `False` otherwise.

Errors: None.

See also: TOrderedList.Count (139)

7.28.7 TOrderedList.Push

Synopsis: Push another element on the list.

Declaration: `function Push(AItem: Pointer) : Pointer`

Visibility: public

Description: `Push` adds `AItem` to the list, and returns `AItem`.

Errors: If not enough memory is available to expand the list, an exception may be raised.

See also: TOrderedList.Pop (140), TOrderedList.Peek (140)

7.28.8 TOrderedList.Pop

Synopsis: Remove an element from the list.

Declaration: `function Pop : Pointer`

Visibility: public

Description: `Pop` removes an element from the list, and returns the element that was removed from the list. If no element is on the list, `Nil` is returned.

Errors: None.

See also: TOrderedList.Peek (140), TOrderedList.Push (139)

7.28.9 TOrderedList.Peek

Synopsis: Return the next element to be popped from the list.

Declaration: `function Peek : Pointer`

Visibility: public

Description: `Peek` returns the element that will be popped from the list at the next call to Pop (140), without actually popping it from the list.

Errors: None.

See also: TOrderedList.Pop (140), TOrderedList.Push (139)

7.29 TQueue

7.29.1 Description

`TQueue` is a descendent of TOrderedList (138) which implements Push (139) and Pop (140) behaviour as a queue: what is first pushed on the queue, is popped of first (FIFO: First in, first out).

`TQueue` offers no new methods, it merely implements some abstract methods introduced by TOrderedList (138)

7.30 TStack

7.30.1 Description

`TStack` is a descendent of TOrderedList (138) which implements Push (139) and Pop (140) behaviour as a stack: what is last pushed on the stack, is popped of first (LIFO: Last in, first out).

`TStack` offers no new methods, it merely implements some abstract methods introduced by TOrderedList (138)

Chapter 8

Reference for unit 'CustApp'

8.1 Used units

Table 8.1: Used units by unit 'CustApp'

Name	Page
Classes	??
sysutils	??

8.2 Overview

The CustApp unit implements the TCustomApplication (142) class, which serves as the common ancestor to many kinds of `TApplication` classes: a GUI application in the LCL, a CGI application in FPCGI, a daemon application in daemonapp. It introduces some properties to describe the environment in which the application is running (environment variables, program command-line parameters) and introduces some methods to initialize and run a program, as well as functionality to handle exceptions.

Typical use of a descendent class is to introduce a global variable `Application` and use the following code:

```
Application.Initialize;
  Application.Run;
```

Since normally only a single instance of this class is created, and it is a `TComponent` descendent, it can be used as an owner for many components, doing so will ensure these components will be freed when the application terminates.

8.3 Constants, types and variables

8.3.1 Types

```
TExceptionEvent = procedure(Sender: TObject;E: Exception) of object
```

`TExceptionEvent` is the prototype for the exception handling events in `TCustomApplication`.

8.4 TCustomApplication

8.4.1 Description

`TCustomApplication` is the ancestor class for classes that whish to implement a global application class instance. It introduces several application-wide functionalities.

- Exception handling in HandleException (143), ShowException (144), OnException (148) and StopOnException (150).
- Command-line parameter parsing in FindOptionIndex (144), GetOptionValue (145), CheckOptions (146) and HasOption (145)
- Environment variable handling in GetEnvironmentList (147) and EnvironmentVariable (149).

Descendent classes need to override the `DoRun` protected method to implement the functionality of the program.

8.4.2 Method overview

Page	Property	Description
146	CheckOptions	Check whether all given options on the command-line are valid.
142	Create	Create a new instance of the `TCustomApplication` class
143	Destroy	Destroys the `TCustomApplication` instance.
144	FindOptionIndex	Return the index of an option.
147	GetEnvironmentList	Return a list of environment variables.
145	GetOptionValue	Return the value of a command-line option.
143	HandleException	Handle an exception.
145	HasOption	Check whether an option was specified.
143	Initialize	Initialize the application
144	Run	Runs the application.
144	ShowException	Show an exception to the user
144	Terminate	Terminate the application.

8.4.3 Property overview

Page	Property	Access	Description
150	CaseSensitiveOptions	rw	Are options interpreted case sensitive or not
148	ConsoleApplication	r	Is the application a console application or not
149	EnvironmentVariable	r	Environment variable access
147	ExeName	r	Name of the executable.
147	HelpFile	rw	Location of the application help file.
148	Location	r	Application location
148	OnException	rw	Exception handling event
150	OptionChar	rw	Command-line switch character
149	ParamCount	r	Number of command-line parameters
149	Params	r	Command-line parameters
150	StopOnException	rw	Should the program loop stop on an exception
147	Terminated	r	Was `Terminate` called or not
148	Title	rw	Application title

8.4.4 TCustomApplication.Create

Synopsis: Create a new instance of the `TCustomApplication` class

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` creates a new instance of the `TCustomApplication` class. It sets some defaults for the various properties, and then calls the inherited `Create`.

See also: TCustomApplication.Destroy (143)

8.4.5 TCustomApplication.Destroy

Synopsis: Destroys the `TCustomApplication` instance.

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` simply calls the inherited `Destroy`.

See also: TCustomApplication.Create (142)

8.4.6 TCustomApplication.HandleException

Synopsis: Handle an exception.

Declaration: `procedure HandleException(Sender: TObject); Virtual`

Visibility: public

Description: `HandleException` is called (or can be called) to handle the exception `Sender`. If the exception is not of class `Exception` then the default handling of exceptions in the SysUtils unit is called.

If the exception is of class `Exception` and the OnException (148) handler is set, the handler is called with the exception object and `Sender` argument.

If the `OnException` handler is not set, then the exception is passed to the ShowException (144) routine, which can be overridden by descendent application classes to show the exception in a way that is fit for the particular class of application. (a GUI application might show the exception in a message dialog.

When the exception is handled in the above manner, and the StopOnException (150) property is set to `True`, the Terminated (147) property is set to `True`, which will cause the Run (144) loop to stop, and the application will exit.

See also: TCustomApplication.ShowException (144), TCustomApplication.StopOnException (150), TCustomApplication.Terminated (147), TCustomApplication.Run (144)

8.4.7 TCustomApplication.Initialize

Synopsis: Initialize the application

Declaration: `procedure Initialize; Virtual`

Visibility: public

Description: `Initialize` can be overridden by descendent applications to perform any initialization after the class was created. It can be used to react to properties being set at program startup. End-user code should call `Initialize` prior to calling `Run`

In `TCustomApplication`, `Initialize` sets `Terminated` to `False`.

See also: TCustomApplication.Run (144), TCustomApplication.Terminated (147)

8.4.8 TCustomApplication.Run

Synopsis: Runs the application.

Declaration: `procedure Run`

Visibility: public

Description: `Run` is the start of the user code: when called, it starts a loop and repeatedly calls `DoRun` until `Terminated` is set to `True`. If an exception is raised during the execution of `DoRun`, it is caught and handled to TCustomApplication.HandleException (143). If TCustomApplication.StopOnException (150) is set to `True` (which is *not* the default), `Run` will exit, and the application will then terminate. The default is to call `DoRun` again, which is useful for applications running a message loop such as services and GUI applications.

See also: TCustomApplication.HandleException (143), TCustomApplication.StopException (142)

8.4.9 TCustomApplication.ShowException

Synopsis: Show an exception to the user

Declaration: `procedure ShowException(E: Exception);  Virtual`

Visibility: public

Description: `ShowException` should be overridden by descendent classes to show an exception message to the user. The default behaviour is to call the ShowException (**??**) procedure in the SysUtils unit.

Descendent classes should do something appropriate for their context: GUI applications can show a message box, daemon applications can write the exception message to the system log, web applications can send a 500 error response code.

Errors: None.

See also: #rtl.sysutils.ShowException (**??**), TCustomApplication.HandleException (143), TCustomApplication.StopException (142)

8.4.10 TCustomApplication.Terminate

Synopsis: Terminate the application.

Declaration: `procedure Terminate;  Virtual`

Visibility: public

Description: `Terminate` sets the `Terminated` property to `True`. By itself, this does not terminate the application. Instead, descendent classes should in their `DoRun` method, check the value of the Terminated (147) property and properly shut down the application if it is set to `True`.

See also: TCustomApplication.Terminated (147), TCustomApplication.Run (144)

8.4.11 TCustomApplication.FindOptionIndex

Synopsis: Return the index of an option.

Declaration: `function FindOptionIndex(const S: String;var Longopt: Boolean) : Integer`

Visibility: public

Description: `FindOptionIndex` will return the index of the option `S` or the long option `LongOpt`. Neither of them should include the switch character. If no such option was specified, -1 is returned. If either the long or short option was specified, then the position on the command-line is returned.

Depending on the value of the CaseSensitiveOptions (150) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with OptionChar (150) (by default the dash ('-') character).

See also: TCustomApplication.HasOption (145), TCustomApplication.GetOptionValue (145), TCustomApplication.CheckOptions (146), TCustomApplication.CaseSensitiveOptions (150), TCustomApplication.OptionChar (150)

8.4.12 TCustomApplication.GetOptionValue

Synopsis: Return the value of a command-line option.

Declaration:
```
function GetOptionValue(const S: String) : String
function GetOptionValue(const C: Char;const S: String) : String
```

Visibility: public

Description: `GetOptionValue` returns the value of an option. Values are specified in the usual GNU option format, either of

```
--longopt=Value
```

or

```
-c Value
```

is supported.

The function returns the specified value, or the empty string if none was specified.

Depending on the value of the CaseSensitiveOptions (150) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with OptionChar (150) (by default the dash ('-') character).

See also: TCustomApplication.FindOptionIndex (144), TCustomApplication.HasOption (145), TCustomApplication.CheckOptions (146), TCustomApplication.CaseSensitiveOptions (150), TCustomApplication.OptionChar (150)

8.4.13 TCustomApplication.HasOption

Synopsis: Check whether an option was specified.

Declaration:
```
function HasOption(const S: String) : Boolean
function HasOption(const C: Char;const S: String) : Boolean
```

Visibility: public

Description: `HasOption` returns `True` if the specified option was given on the command line. Either the short option character `C` or the long option `S` may be used. Note that both options (requiring a value) and switches can be specified.

Depending on the value of the CaseSensitiveOptions (150) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with OptionChar (150) (by default the dash ('-') character).

See also: TCustomApplication.FindOptionIndex (144), TCustomApplication.GetOptionValue (145), TCustomApplication.CheckOptions (146), TCustomApplication.CaseSensitiveOptions (150), TCustomApplication.OptionChar (150)

8.4.14 TCustomApplication.CheckOptions

Synopsis: Check whether all given options on the command-line are valid.

Declaration:
```
function CheckOptions(const ShortOptions: String;
                      const Longopts: TStrings;Opts: TStrings;
                      NonOpts: TStrings) : String
function CheckOptions(const ShortOptions: String;
                      const Longopts: TStrings) : String
function CheckOptions(const ShortOptions: String;
                      const LongOpts: Array of String) : String
function CheckOptions(const ShortOptions: String;const LongOpts: String)
                       : String
```

Visibility: public

Description: `CheckOptions` scans the command-line and checks whether the options given are valid options. It also checks whether options that require a valued are indeed specified with a value.

The `ShortOptions` contains a string with valid short option characters. Each character in the string is a valid option character. If a character is followed by a colon (:), then a value must be specified. If it is followed by 2 colon characters (::) then the value is optional.

`LongOpts` is a list of strings (which can be specified as an array, a `TStrings` instance or a string with whitespace-separated values) of valid long options.

When the function returns, if `Opts` is non-`Nil`, the `Opts` stringlist is filled with the passed valid options. If `NonOpts` is non-nil, it is filled with any non-option strings that were passed on the command-line.

The function returns an empty string if all specified options were valid options, and whether options requiring a value have a value. If an error was found during the check, the return value is a string describing the error.

Options are identified as command-line parameters which start with OptionChar (150) (by default the dash ('-') character).

Errors: if an error was found during the check, the return value is a string describing the error.

See also: TCustomApplication.FindOptionIndex (144), TCustomApplication.GetOptionValue (145), TCustomApplication.HasOption (145), TCustomApplication.CaseSensitiveOptions (150), TCustomApplication.OptionChar (150)

8.4.15 TCustomApplication.GetEnvironmentList

Synopsis: Return a list of environment variables.

Declaration:
```
procedure GetEnvironmentList(List: TStrings;NamesOnly: Boolean)
procedure GetEnvironmentList(List: TStrings)
```

Visibility: public

Description: `GetEnvironmentList` returns a list of environment variables in `List`. They are in the form `Name=Value`, one per item in `list`. If `NamesOnly` is `True`, then only the names are returned.

See also: TCustomApplication.EnvironmentVariable (149)

8.4.16 TCustomApplication.ExeName

Synopsis: Name of the executable.

Declaration:
```
Property ExeName : String
```

Visibility: public

Access: Read

Description: `ExeName` returns the full name of the executable binary (path+filename). This is equivalent to `Paramstr(0)`

Note that some operating systems do not return the full pathname of the binary.

See also: #rtl.system.paramstr (**??**)

8.4.17 TCustomApplication.HelpFile

Synopsis: Location of the application help file.

Declaration:
```
Property HelpFile : String
```

Visibility: public

Access: Read,Write

Description: `HelpFile` is the location of the application help file. It is a simple string property which can be set by an IDE such as Lazarus, and is mainly provided for compatibility with Delphi's `TApplication` implementation.

See also: TCustomApplication.Title (148)

8.4.18 TCustomApplication.Terminated

Synopsis: Was `Terminate` called or not

Declaration:
```
Property Terminated : Boolean
```

Visibility: public

Access: Read

Description: `Terminated` indicates whether Terminate (144) was called or not. Descendent classes should check `Terminated` at regular intervals in their implementation of `DoRun`, and if it is set to `True`, should exit gracefully the `DoRun` method.

See also: TCustomApplication.Terminate (144)

8.4.19 TCustomApplication.Title

Synopsis: Application title

Declaration: `Property Title : String`

Visibility: public

Access: Read,Write

Description: `Title` is a simple string property which can be set to any string describing the application. It does nothing by itself, and is mainly introduced for compatibility with Delphi's `TApplication` implementation.

See also: TCustomApplication.HelpFile (147)

8.4.20 TCustomApplication.OnException

Synopsis: Exception handling event

Declaration: `Property OnException : TExceptionEvent`

Visibility: public

Access: Read,Write

Description: `OnException` can be set to provide custom handling of events, instead of the default action, which is simply to show the event using ShowEvent (142).

If set, `OnException` is called by the HandleEvent (142) routine. Do not use the `OnException` event directly, instead call `HandleEvent`

See also: TCustomApplication.ShowEvent (142)

8.4.21 TCustomApplication.ConsoleApplication

Synopsis: Is the application a console application or not

Declaration: `Property ConsoleApplication : Boolean`

Visibility: public

Access: Read

Description: `ConsoleApplication` returns `True` if the application is compiled as a console application (the default) or `False` if not. The result of this property is determined at compile-time by the settings of the compiler: it returns the value of the IsConsole (**??**) constant.

See also: #rtl.system.IsConsole (**??**)

8.4.22 TCustomApplication.Location

Synopsis: Application location

Declaration: `Property Location : String`

Visibility: public

Access: Read

Description: `Location` returns the directory part of the application binary. This property works on most platforms, although some platforms do not allow to retrieve this information (Mac OS under certain circumstances). See the discussion of Paramstr (**??**) in the RTL documentation.

See also: #rtl.system.paramstr (**??**), TCustomApplication.Params (149)

8.4.23 TCustomApplication.Params

Synopsis: Command-line parameters

Declaration: `Property Params[Index: Integer]: String`

Visibility: public

Access: Read

Description: `Params` gives access to the command-line parameters. They contain the value of the `Index`-th parameter, where `Index` runs from 0 to ParamCount (149). It is equivalent to calling ParamStr (**??**).

See also: TCustomApplication.ParamCount (149), #rtl.system.paramstr (**??**)

8.4.24 TCustomApplication.ParamCount

Synopsis: Number of command-line parameters

Declaration: `Property ParamCount : Integer`

Visibility: public

Access: Read

Description: `ParamCount` returns the number of command-line parameters that were passed to the program. The actual parameters can be retrieved with the Params (149) property.

See also: TCustomApplication.Params (149), #rtl.system.paramstr (**??**), #rtl.system.paramcount (**??**)

8.4.25 TCustomApplication.EnvironmentVariable

Synopsis: Environment variable access

Declaration: `Property EnvironmentVariable[envName: String]: String`

Visibility: public

Access: Read

Description: `EnvironmentVariable` gives access to the environment variables of the application: It returns the value of the environment variable `EnvName`, or an empty string if no such value is available.

To use this property, the name of the environment variable must be known. To get a list of available names (and values), GetEnvironmentList (147) can be used.

See also: TCustomApplication.GetEnvironmentList (147), TCustomApplication.Params (149)

8.4.26 TCustomApplication.OptionChar

Synopsis: Command-line switch character

Declaration: `Property OptionChar : Char`

Visibility: public

Access: Read,Write

Description: `OptionChar` is the character used for command line switches. By default, this is the dash ('-') character, but it can be set to any other non-alphanumerical character (although no check is performed on this).

See also: TCustomApplication.FindOptionIndex (144), TCustomApplication.GetOptionValue (145), TCustomApplication.HasOption (145), TCustomApplication.CaseSensitiveOptions (150), TCustomApplication.CheckOptions (146)

8.4.27 TCustomApplication.CaseSensitiveOptions

Synopsis: Are options interpreted case sensitive or not

Declaration: `Property CaseSensitiveOptions : Boolean`

Visibility: public

Access: Read,Write

Description: `CaseSensitiveOptions` determines whether FindOptionIndex (144) and CheckOptions (146) perform searches in a case sensitive manner or not. By default, the search is case-sensitive. Setting this property to `False` makes the search case-insensitive.

See also: TCustomApplication.FindOptionIndex (144), TCustomApplication.GetOptionValue (145), TCustomApplication.HasOption (145), TCustomApplication.OptionChar (150), TCustomApplication.CheckOptions (146)

8.4.28 TCustomApplication.StopOnException

Synopsis: Should the program loop stop on an exception

Declaration: `Property StopOnException : Boolean`

Visibility: public

Access: Read,Write

Description: `StopOnException` controls the behaviour of the Run (144) and HandleException (143) procedures in case of an unhandled exception in the `DoRun` code. If `StopOnException` is `True` then Terminate (144) will be called after the exception was handled.

See also: TCustomApplication.Run (144), TCustomApplication.HandleException (143), TCustomApplication.Terminate (144)

Chapter 9

Reference for unit 'daemonapp'

9.1 Daemon application architecture

[Still needs to be completed]

9.2 Used units

Table 9.1: Used units by unit 'daemonapp'

Name	Page
Classes	??
CustApp	141
eventlog	200
rtlconsts	151
sysutils	??

9.3 Overview

The daemonapp unit implements a `TApplication` class which encapsulates a daemon or service application. It handles installation where this is necessary, and does instantiation of the various daemons where necessary.

The unit consists of 3 separate classes which cooperate tightly:

TDaemon This is a class that implements the daemon's functionality. One or more descendents of this class can be implemented and instantiated in a single daemon application. For more information, see TDaemon (166).

TDaemonApplication This is the actual daemon application class. A global instance of this class is instantiated. It handles the command-line arguments, and instantiates the various daemons. For more information, see TDaemonApplication (171).

TDaemonDef This class defines the daemon in the operation system. The `TDaemonApplication` class has a collection of `TDaemonDef` instances, which it uses to start the various daemons. For more information, see TDaemonDef (174).

As can be seen, a single application can implement one ore more daemons (services). Each daemon will be run in a separate thread which is controlled by the application class.

The classes take care of logging through the TEventLog (202) class.

Many options are needed only to make the application behave as a windows service application on windows. These options are ignored in unix-like environment. The documentation will mention this.

9.4 Constants, types and variables

9.4.1 Resource strings

```
SControlFailed = 'Control code %s handling failed: %s'
```

The control code was not handled correctly

```
SCustomCode = '[Custom code %d]'
```

A custom code was received

```
SDaemonStatus = 'Daemon %s current status: %s'
```

Daemon status report log message

```
SErrApplicationAlreadyCreated = 'An application instance of class %s was already cre
```

A second application instance is created

```
SErrDaemonStartFailed = 'Failed to start daemon %s : %s'
```

The application failed to start the daemon

```
SErrDuplicateName = 'Duplicate daemon name: %s'
```

Duplicate service name

```
SErrNoDaemonDefForStatus = '%s: No daemon definition for status report'
```

Internal error: no daemon definition to report status for

```
SErrNoDaemonForStatus = '%s: No daemon for status report'
```

Internal error: no daemon to report status for

```
SErrNoServiceMapper = 'No daemon mapper class registered.'
```

No service mapper was found.

```
SErrNothingToDo = 'Options do not allow determining what needs to be done.'
```

No operation can be performed

```
SErrOnlyOneMapperAllowed = 'Not changing daemon mapper class %s with %s: Only 1 mapp
```

An attempt was made to install a second service mapper

```
SErrServiceManagerStartFailed = 'Failed to start service manager: %s'
```

Unable to start or contact the service manager

```
SErrUnknownDaemonClass = 'Unknown daemon class name: %s'
```

Unknown daemon class requested

```
SErrWindowClass = 'Could not register window class'
```

Could not register window class

9.4.2 Types

```
TCurrentStatus = (csStopped,csStartPending,csStopPending,csRunning,
                  csContinuePending,csPausePending,csPaused)
```

Table 9.2: Enumeration values for type TCurrentStatus

Value	Explanation
csContinuePending	The daemon is continuing, but not yet running
csPaused	The daemon is paused: running but not active.
csPausePending	The daemon is about to be paused.
csRunning	The daemon is running (it is operational).
csStartPending	The daemon is starting, but not yet fully running.
csStopped	The daemon is stopped, i.e. inactive.
csStopPending	The daemon is stopping, but not yet fully stopped.

`TCurrentStatus` indicates the current state of the daemon. It changes from one state to the next during the time the instance is active. The daemon application changes the state of the daemon, depending on signals it gets from the operating system, by calling the appropriate methods.

```
TCustomControlCodeEvent = procedure(Sender: TCustomDaemon;ACode: DWord;
                                    var Handled: Boolean) of object
```

In case the system sends a non-standard control code to the daemon, an event handler is executed with this prototype.

```
TCustomDaemonApplicationClass = Class of TCustomDaemonApplication
```

Class pointer for `TCustomDaemonApplication`

```
TCustomDaemonClass = Class of TCustomDaemon
```

The class type is needed in the TDaemonDef (174) definition.

```
TCustomDaemonMapperClass = Class of TCustomDaemonMapper
```

`TCustomDaemonMapperClass` is the class of `TCustomDaemonMapper`. It is used in the RegisterDaemonMapper (157) call.

```
TDaemonClass = Class of TDaemon
```

Class type of `TDaemon`

```
TDaemonEvent = procedure(Sender: TCustomDaemon) of object
```

`TDaemonEvent` is used in event handling. The `Sender` is the TCustomDaemon (157) instance that has initiated the event.

```
TDaemonOKEvent = procedure(Sender: TCustomDaemon;var OK: Boolean)
                           of object
```

`TDaemonOKEvent` is used in event handling, when a boolean result must be obtained, for instance, to see if an operation was performed succesfully.

```
TDaemonOption = (doAllowStop,doAllowPause,doInteractive)
```

Table 9.3: Enumeration values for type TDaemonOption

Value	Explanation
doAllowPause	The daemon can be paused.
doAllowStop	The daemon can be stopped.
doInteractive	The daemon interacts with the desktop.

Enumerated that enumerates the various daemon operation options.

```
TDaemonOptions= Set of (doAllowPause,doAllowStop,doInteractive)
```

`TDaemonOption` enumerates the various options a daemon can have.

```
TDaemonRunMode = (drmUnknown,drmInstall,drmUninstall,drmRun)
```

Table 9.4: Enumeration values for type TDaemonRunMode

Value	Explanation
drmInstall	Daemon install mode (windows only)
drmRun	Daemon is running normally
drmUninstall	Daemon uninstall mode (windows only)
drmUnknown	Unknown mode

`TDaemonRunMode` indicates in what mode the daemon application (as a whole) is currently running.

```
TErrorSeverity = (esIgnore,esNormal,esSevere,esCritical)
```

`TErrorSeverity` determines what action windows takes when the daemon fails to start. It is used on windows only, and is ignored on other platforms.

Table 9.5: Enumeration values for type TErrorSeverity

Value	Explanation
esCritical	Error is logged, and startup is stopped if last known good configuration is active, or system is restarted using last
esIgnore	Ignore startup errors
esNormal	Error is logged, but startup continues
esSevere	Error is logged, and startup is continued if last known good configuration is active, or system is restarted using la

```
TGuiLoopEvent = procedure of object
```

`TGuiLoopEvent` is the main GUI loop event procedure prototype. It is called by the application instance in case the daemon has a visual part, which needs to handle visual events. It is run in the main application thread.

```
TServiceType = (stWin32,stDevice,stFileSystem)
```

Table 9.6: Enumeration values for type TServiceType

Value	Explanation
stDevice	Device driver
stFileSystem	File system driver
stWin32	Regular win32 service

The type of service. This type is used on windows only, to signal the operating system what kind of service is being installed or run.

```
TStartType = (stBoot,stSystem,stAuto,stManual,stDisabled)
```

Table 9.7: Enumeration values for type TStartType

Value	Explanation
stAuto	Started automatically by service manager during system startup
stBoot	During system boot
stDisabled	Service is not started, it is disabled
stManual	Started manually by the user or other processes.
stSystem	During load of device drivers

`TStartType` can be used to define when the service must be started on windows. This type is not used on other platforms.

9.4.3 Variables

```
CurrentStatusNames : Array[TCurrentStatus] of String = ('Stopped','Start Pending','S
```

Names for various service statuses

```
DefaultDaemonOptions : TDaemonOptions = [doAllowStop,doAllowPause]
```

`DefaultDaemonOptions` are the default options with which a daemon definition (TDaemonDef (174)) is created.

```
SStatus : Array[1..5] of String = ('Stop','Pause','Continue','Interrogate','Shutdown
```

Status message

9.5 Procedures and functions

9.5.1 Application

Synopsis: Application instance

Declaration: `function Application : TCustomDaemonApplication`

Visibility: default

Description: `Application` is the TCustomDaemonApplication (160) instance used by this application. The instance is created at the first invocation of this function, so it is possible to use RegisterDaemonApplicationClass (156) to register an alternative `TCustomDaemonApplication` class to run the application.

See also: TCustomDaemonApplication (160), RegisterDaemonApplicationClass (156)

9.5.2 DaemonError

Synopsis: Raise an `EDaemon` exception

Declaration:
```
procedure DaemonError(Msg: String)
procedure DaemonError(Fmt: String;Args: Array of const)
```

Visibility: default

Description: `DaemonError` raises an EDaemon (157) exception with message `Msg` or it formats the message using `Fmt` and `Args`.

See also: EDaemon (157)

9.5.3 RegisterDaemonApplicationClass

Synopsis: Register alternative `TCustomDaemonApplication` class.

Declaration:
```
procedure RegisterDaemonApplicationClass
                                        (AClass: TCustomDaemonApplicationClass)
```

Visibility: default

Description: `RegisterDaemonApplicationClass` can be used to register an alternative TCustomDaemonApplication (160) descendent which will be used when creating the global Application (156) instance. Only the last registered class pointer will be used.

See also: TCustomDaemonApplication (160), Application (156)

9.5.4 RegisterDaemonClass

Synopsis: Register daemon

Declaration: `procedure RegisterDaemonClass(AClass: TCustomDaemonClass)`

Visibility: default

Description: `RegisterDaemonClass` must be called for each TCustomDaemon (157) descendent that is used in the class: the class pointer and class name are used by the TCustomDaemonMapperClass (153) class to create a `TCustomDaemon` instance when a daemon is required.

See also: TCustomDaemonMapperClass (153), TCustomDaemon (157)

9.5.5 RegisterDaemonMapper

Synopsis: Register a daemon mapper class

Declaration: `procedure RegisterDaemonMapper(AMapperClass: TCustomDaemonMapperClass)`

Visibility: default

Description: `RegisterDaemonMapper` can be used to register an alternative class for the global daemonmapper. The daemonmapper will be used only when the application is being run, by the TCustomDaemonApplication (160) code, so registering an alternative mapping class should happen in the initialization section of the application units.

See also: TCustomDaemonApplication (160), TCustomDaemonMapperClass (153)

9.6 EDaemon

9.6.1 Description

`EDaemon` is the exception class used by all code in the DaemonApp unit.

9.7 TCustomDaemon

9.7.1 Description

`TCustomDaemon` implements all the basic calls that are needed for a daemon to function. Descendents of `TCustomDaemon` can override these calls to implement the daemon-specific behaviour.

`TCustomDaemon` is an abstract class, it should never be instantiated. Either a descendent of it must be created and instantiated, or a descendent of TDaemon (166) can be designed to implement the behaviour of the daemon.

9.7.2 Method overview

Page	Property	Description
158	LogMessage	Log a message to the system log
158	ReportStatus	Report the current status to the operating system

9.7.3 Property overview

Page	Property	Access	Description
159	Controller	r	`TDaemonController` instance controlling this daemon instance
159	DaemonThread	r	Thread in which daemon is running
158	Definition	r	The definition used to instantiate this daemon instance
160	Logger	r	`TEventLog` instance used to send messages to the system log
159	Status	rw	Current status of the daemon

9.7.4 TCustomDaemon.LogMessage

Synopsis: Log a message to the system log

Declaration: `procedure LogMessage(Msg: String)`

Visibility: public

Description: `LogMessage` can be used to send a message `Msg` to the system log. A TEventLog (202) instance is used to actually send messages to the system log.

The message is sent with an 'error' flag (using TEventLog.Error (205)).

Errors: None.

See also: TCustomDaemon.ReportStatus (158)

9.7.5 TCustomDaemon.ReportStatus

Synopsis: Report the current status to the operating system

Declaration: `procedure ReportStatus`

Visibility: public

Description: `ReportStatus` can be used to report the current status to the operating system. The start and stop or pause and continue operations can be slow to start up. This call can (and should) be used to report the current status to the operating system during such lengthy operations, or else it may conclude that the daemon has died.

This call is mostly important on windows operating systems, to notify the service manager that the operation is still in progress.

The implementation of `ReportStatus` simply calls `ReportStatus` in the controller.

Errors: None.

See also: TCustomDaemon.LogMessage (158)

9.7.6 TCustomDaemon.Definition

Synopsis: The definition used to instantiate this daemon instance

Declaration: `Property Definition : TDaemonDef`

Visibility: public

Access: Read

Description: `Definition` is the TDaemonDef (174) definition that was used to start the daemon instance. It can be used to retrieve additional information about the intended behaviour of the daemon.

See also: TDaemonDef (174)

9.7.7 TCustomDaemon.DaemonThread

Synopsis: Thread in which daemon is running

Declaration: `Property DaemonThread : TThread`

Visibility: public

Access: Read

Description: `DaemonThread` is the thread in which the daemon instance is running. Each daemon instance in the application runs in it's own thread, none of which are the main thread of the application. The application main thread is used to handle control messages coming from the operating system.

See also: TCustomDaemon.Controller (159)

9.7.8 TCustomDaemon.Controller

Synopsis: `TDaemonController` instance controlling this daemon instance

Declaration: `Property Controller : TDaemonController`

Visibility: public

Access: Read

Description: `Controller` points to the `TDaemonController` instance that was created by the application instance to control this daemon.

See also: TCustomDaemon.DaemonThread (159)

9.7.9 TCustomDaemon.Status

Synopsis: Current status of the daemon

Declaration: `Property Status : TCurrentStatus`

Visibility: public

Access: Read,Write

Description: `Status` indicates the current status of the daemon. It is set by the various operations that the controller operates on the daemon, and should not be set manually.

`Status` is the value which `ReportStatus` will send to the operating system.

See also: TCustomDaemon.ReportStatus (158)

9.7.10 TCustomDaemon.Logger

Synopsis: `TEventLog` instance used to send messages to the system log

Declaration: `Property Logger : TEventLog`

Visibility: public

Access: Read

Description: `Logger` is the TEventLog (202) instance used to send messages to the system log. It is used by the LogMessage (158) call, but is acessible through the `Logger` property in case more configurable logging is needed than offered by `LogMessage`.

See also: TCustomDaemon.LogMessage (158), #fcl.eventlog.TEventLog (202)

9.8 TCustomDaemonApplication

9.8.1 Description

`TCustomDaemonApplication` is a TCustomApplication (142) descendent which is the main application instance for a daemon. It handles the command-line and decides what to do when the application is started, depending on the command-line options given to the application, by calling the various methods.

It creates the necessary TDaemon (166) instances by checking the TCustomDaemonMapperClass (153) instance that contains the daemon maps.

9.8.2 Method overview

Page	Property	Description
161	CreateDaemon	Create daemon instance
162	CreateForm	Create a component
161	InstallDaemons	Install all daemons.
161	RunDaemons	Run all daemons.
160	ShowException	Show an exception
161	StopDaemons	Stop all daemons
162	UnInstallDaemons	Uninstall all daemons

9.8.3 Property overview

Page	Property	Access	Description
163	GuiHandle	rw	Handle of GUI loop main application window handle
163	GUIMainLoop	rw	GUI main loop callback
162	Logger	r	Event logging instance used for logging messages
163	RunMode	r	Application mode

9.8.4 TCustomDaemonApplication.ShowException

Synopsis: Show an exception

Declaration: `procedure ShowException(E: Exception);  Override`

Visibility: public

Description: `ShowException` is overridden by `TCustomDaemonApplication`, it sends the exception message to the system log.

9.8.5 TCustomDaemonApplication.CreateDaemon

Synopsis: Create daemon instance

Declaration: `function CreateDaemon(DaemonDef: TDaemonDef) : TCustomDaemon`

Visibility: public

Description: `CreateDaemon` is called whenever a TCustomDaemon (157) instance must be created from a TDaemonDef (174) daemon definition, passed in `DaemonDef`. It initializes the `TCustomDaemon` instance, and creates a controller instance of type TDaemonController (171) to control the daemon. Finally, it assigns the created daemon to the TDaemonDef.Instance (175) property.

Errors: In case of an error, an exception may be raised.

See also: TDaemonController (171), TCustomDaemon (157), TDaemonDef (174), TDaemonDef.Instance (175)

9.8.6 TCustomDaemonApplication.StopDaemons

Synopsis: Stop all daemons

Declaration: `procedure StopDaemons(Force: Boolean)`

Visibility: public

Description: `StopDaemons` sends the `STOP` control code to all daemons, or the `SHUTDOWN` control code in case `Force` is `True`.

See also: TDaemonController.Controller (172), TCustomDaemonApplication.UnInstallDaemons (162), TCustomDaemonApplication.RunDaemons (161)

9.8.7 TCustomDaemonApplication.InstallDaemons

Synopsis: Install all daemons.

Declaration: `procedure InstallDaemons`

Visibility: public

Description: `InstallDaemons` installs all known daemons, i.e. registers them with the service manager on Windows. This method is called if the application is run with the `-i` or `-install` or `/install` command-line option.

See also: TCustomDaemonApplication.UnInstallDaemons (162), TCustomDaemonApplication.RunDaemons (161), TCustomDaemonApplication.StopDaemons (161)

9.8.8 TCustomDaemonApplication.RunDaemons

Synopsis: Run all daemons.

Declaration: `procedure RunDaemons`

Visibility: public

Description: `RunDaemons` runs (starts) all known daemons. This method is called if the application is run with the `-r` or `-run` methods.

Errors:

See also: TCustomDaemonApplication.UnInstallDaemons (162), TCustomDaemonApplication.InstallDaemons (161), TCustomDaemonApplication.StopDaemons (161)

9.8.9 TCustomDaemonApplication.UnInstallDaemons

Synopsis: Uninstall all daemons

Declaration:
```
procedure UnInstallDaemons
```

Visibility: public

Description: `UnInstallDaemons` uninstalls all known daemons, i.e. deregisters them with the service manager on Windows. This method is called if the application is run with the `-u` or `-uninstall` or `/uninstall` command-line option.

See also: TCustomDaemonApplication.RunDaemons (161), TCustomDaemonApplication.InstallDaemons (161), TCustomDaemonApplication.StopDaemons (161)

9.8.10 TCustomDaemonApplication.CreateForm

Synopsis: Create a component

Declaration:
```
procedure CreateForm(InstanceClass: TComponentClass;var Reference)
                          ;  Virtual
```

Visibility: public

Description: `CreateForm` creates an instance of `InstanceClass` and fills `Reference` with the class instance pointer. It's main purpose is to give an IDE a means of assuring that forms or datamodules are created on application startup: the IDE will generate calls for all modules that are auto-created.

Errors: An exception may arise if the instance wants to stream itself from resources, but no resources are found.

See also: TCustomDaemonApplication.CreateDaemon (161)

9.8.11 TCustomDaemonApplication.Logger

Synopsis: Event logging instance used for logging messages

Declaration:
```
Property Logger : TEventLog
```

Visibility: public

Access: Read

Description: `Logger` contains a reference to the TEventLog (202) instance that can be used to send messages to the system log.

See also: TCustomDaemon.LogMessage (158)

9.8.12 TCustomDaemonApplication.GUIMainLoop

Synopsis: GUI main loop callback

Declaration: `Property GUIMainLoop : TGuiLoopEvent`

Visibility: public

Access: Read,Write

Description: `GUIMainLoop` contains a reference to a method that can be called to process a main GUI loop. The procedure should return only when the main GUI has finished and the application should exit. It is called when the daemons are running.

See also: TCustomDaemonApplication.GuiHandle (163)

9.8.13 TCustomDaemonApplication.GuiHandle

Synopsis: Handle of GUI loop main application window handle

Declaration: `Property GuiHandle : THandle`

Visibility: public

Access: Read,Write

Description: `GuiHandle` is the handle of a GUI window which can be used to run a message handling loop on. It is created when no GUIMainLoop (163) procedure exists, and the application creates and runs a message loop by itself.

See also: TCustomDaemonApplication.GUIMainLoop (163)

9.8.14 TCustomDaemonApplication.RunMode

Synopsis: Application mode

Declaration: `Property RunMode : TDaemonRunMode`

Visibility: public

Access: Read

Description: `RunMode` indicates in which mode the application is running currently. It is set automatically by examining the command-line, and when set, one of InstallDaemons (161), RunDaemons (161) or UnInstallDaemons (162) is called.

See also: TCustomDaemonApplication.InstallDaemons (161), TCustomDaemonApplication.RunDaemons (161), TCustomDaemonApplication.UnInstallDaemons (162)

9.9 TCustomDaemonMapper

9.9.1 Description

The `TCustomDaemonMapper` class is responsible for mapping a daemon definition to an actual `TDaemon` instance. It maintains a TDaemonDefs (178) collection with daemon definitions, which can be used to map the definition of a daemon to a `TDaemon` descendent class.

An IDE such as Lazarus can design a `TCustomDaemonMapper` instance visually, to help establish the relationship between various TDaemonDef (174) definitions and the actual TDaemon (166) instances that will be used to run the daemons.

The `TCustomDaemonMapper` class has no support for streaming. The TDaemonMapper (180) class has support for streaming (and hence visual designing).

9.9.2 Method overview

Page	Property	Description
164	Create	Create a new instance of `TCustomDaemonMapper`
164	Destroy	Clean up and destroy a `TCustomDaemonMapper` instance.

9.9.3 Property overview

Page	Property	Access	Description
164	DaemonDefs	rw	Collection of daemons
165	OnCreate	rw	Event called when the daemon mapper is created
165	OnDestroy	rw	Event called when the daemon mapper is freed.
166	OnInstall	rw	Event called when the daemons are installed
165	OnRun	rw	Event called when the daemons are executed.
166	OnUnInstall	rw	Event called when the daemons are uninstalled

9.9.4 TCustomDaemonMapper.Create

Synopsis: Create a new instance of `TCustomDaemonMapper`

Declaration: `constructor Create(AOwner: TComponent);  Override`

Visibility: public

Description: `Create` creates a new instance of a `TCustomDaemonMapper`. It creates the TDaemonDefs (178) collection and then calls the inherited constructor. It should never be necessary to create a daemon mapper manually, the application will create a global `TCustomDaemonMapper` instance.

See also: TDaemonDefs (178), TCustomDaemonApplication (160), TCustomDaemonMapper.Destroy (164)

9.9.5 TCustomDaemonMapper.Destroy

Synopsis: Clean up and destroy a `TCustomDaemonMapper` instance.

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` frees the DaemonDefs (164) collection and calls the inherited destructor.

See also: TDaemonDefs (178), TCustomDaemonMapper.Create (164)

9.9.6 TCustomDaemonMapper.DaemonDefs

Synopsis: Collection of daemons

Declaration: `Property DaemonDefs : TDaemonDefs`

Visibility: published

Access: Read,Write

Description: `DaemonDefs` is the application's global collection of daemon definitions. This collection will be used to decide at runtime which `TDaemon` class must be created to run or install a daemon.

See also: TCustomDaemonApplication (160)

9.9.7 TCustomDaemonMapper.OnCreate

Synopsis: Event called when the daemon mapper is created

Declaration: `Property OnCreate : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnCreate` is an event that is called when the `TCustomDaemonMapper` instance is created. It can for instance be used to dynamically create daemon definitions at runtime.

See also: TCustomDaemonMapper.OnDestroy (165), TCustomDaemonMapper.OnUnInstall (166), TCustomDaemonMapper.OnCreate (165), TCustomDaemonMapper.OnDestroy (165)

9.9.8 TCustomDaemonMapper.OnDestroy

Synopsis: Event called when the daemon mapper is freed.

Declaration: `Property OnDestroy : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnDestroy` is called when the global daemon mapper instance is destroyed. it can be used to release up any resources that were allocated when the instance was created, in the OnCreate (165) event.

See also: TCustomDaemonMapper.OnCreate (165), TCustomDaemonMapper.OnInstall (166), TCustomDaemonMapper.OnUnInstall (166), TCustomDaemonMapper.OnCreate (165)

9.9.9 TCustomDaemonMapper.OnRun

Synopsis: Event called when the daemons are executed.

Declaration: `Property OnRun : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnRun` is the event called when the daemon application is executed to run the daemons (with command-line parameter '-r'). it is called exactly once.

See also: TCustomDaemonMapper.OnInstall (166), TCustomDaemonMapper.OnUnInstall (166), TCustomDaemonMapper.OnCreate (165), TCustomDaemonMapper.OnDestroy (165)

9.9.10 TCustomDaemonMapper.OnInstall

Synopsis: Event called when the daemons are installed

Declaration: `Property OnInstall : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnInstall` is the event called when the daemon application is executed to install the daemons (with command-line parameter '-i' or '/install'). it is called exactly once.

See also: TCustomDaemonMapper.OnRun (165), TCustomDaemonMapper.OnUnInstall (166), TCustomDaemonMapper.OnCreate (165), TCustomDaemonMapper.OnDestroy (165)

9.9.11 TCustomDaemonMapper.OnUnInstall

Synopsis: Event called when the daemons are uninstalled

Declaration: `Property OnUnInstall : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnUnInstall` is the event called when the daemon application is executed to uninstall the daemons (with command-line parameter '-u' or '/uninstall'). it is called exactly once.

See also: TCustomDaemonMapper.OnRun (165), TCustomDaemonMapper.OnInstall (166), TCustomDaemonMapper.OnCreate (165), TCustomDaemonMapper.OnDestroy (165)

9.10 TDaemon

9.10.1 Description

`TDaemon` is a TCustomDaemon (157) descendent which is meant for development in a visual environment: it contains event handlers for all major operations. Whenever a `TCustomDaemon` method is executed, it's execution is shunted to the event handler, which can be filled with code in the IDE.

All the events of the daemon are executed in the thread in which the daemon's controller is running (as given by DaemonThread (159)), which is not the main program thread.

9.10.2 Property overview

Page	Property	Access	Description
170	AfterInstall	rw	Called after the daemon was installed
170	AfterUnInstall	rw	Called after the daemon is uninstalled
169	BeforeInstall	rw	Called before the daemon will be installed
170	BeforeUnInstall	rw	Called before the daemon is uninstalled
167	Definition		
168	OnContinue	rw	Daemon continue
170	OnControlCode	rw	Called when a control code is received for the daemon
169	OnExecute	rw	Daemon execute event
168	OnPause	rw	Daemon pause event
169	OnShutDown	rw	Daemon shutdown
167	OnStart	rw	Daemon start event
168	OnStop	rw	Daemon stop event
167	Status		

9.10.3 TDaemon.Definition

Declaration: `Property Definition :`

Visibility: public

Access:

9.10.4 TDaemon.Status

Declaration: `Property Status :`

Visibility: public

Access:

9.10.5 TDaemon.OnStart

Synopsis: Daemon start event

Declaration: `Property OnStart : TDaemonOKEvent`

Visibility: published

Access: Read,Write

Description: `OnStart` is the event called when the daemon must be started. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the ReportStatus (158) method.

If the start of the daemon should do some continuous action, then this action should be performed in a new thread: this thread should then be created and started in the OnExecute (169) event handler, so the event handler can return at once.

See also: TDaemon.OnStop (168), TDaemon.OnExecute (169), TDaemon.OnContinue (168), TCustomDaemon.ReportStatus (158)

9.10.6 TDaemon.OnStop

Synopsis: Daemon stop event

Declaration: `Property OnStop : TDaemonOKEvent`

Visibility: published

Access: Read,Write

Description: `OnStart` is the event called when the daemon must be stopped. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the ReportStatus (158) method.

If a thread was started in the OnExecute (169) event, this is the place where the thread should be stopped.

See also: TDaemon.OnStart (167), TDaemon.OnPause (168), TCustomDaemon.ReportStatus (158)

9.10.7 TDaemon.OnPause

Synopsis: Daemon pause event

Declaration: `Property OnPause : TDaemonOKEvent`

Visibility: published

Access: Read,Write

Description: `OnPause` is the event called when the daemon must be stopped. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the ReportStatus (158) method.

If a thread was started in the OnExecute (169) event, this is the place where the thread's execution should be suspended.

See also: TDaemon.OnStop (168), TDaemon.OnContinue (168), TCustomDaemon.ReportStatus (158)

9.10.8 TDaemon.OnContinue

Synopsis: Daemon continue

Declaration: `Property OnContinue : TDaemonOKEvent`

Visibility: published

Access: Read,Write

Description: `OnPause` is the event called when the daemon must be stopped. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the ReportStatus (158) method.

If a thread was started in the OnExecute (169) event and it was suspended in a OnPause (167) event, this is the place where the thread's executed should be resumed.

See also: TDaemon.OnStart (167), TDaemon.OnPause (168), TCustomDaemon.ReportStatus (158)

9.10.9 TDaemon.OnShutDown

Synopsis: Daemon shutdown

Declaration: `Property OnShutDown : TDaemonEvent`

Visibility: published

Access: Read,Write

Description: `OnShutDown` is the event called when the daemon must be shut down. When the system is being shut down and the daemon does not respond to stop signals, then a shutdown message is sent to the daemon. This event can be used to respond to such a message. The daemon process will simply be stopped after this event.

If a thread was started in the OnExecute (169), this is the place where the thread's executed should be stopped or the thread freed from memory.

See also: TDaemon.OnStart (167), TDaemon.OnPause (168), TCustomDaemon.ReportStatus (158)

9.10.10 TDaemon.OnExecute

Synopsis: Daemon execute event

Declaration: `Property OnExecute : TDaemonEvent`

Visibility: published

Access: Read,Write

Description: `OnExecute` is executed once after the daemon was started. If assigned, it should perform whatever operation the daemon is designed.

If the daemon's action is event based, then no `OnExecute` handler is needed, and the events will control the daemon's execution: the daemon thread will then go in a loop, passing control messages to the daemon.

If an `OnExecute` event handler is present, the checking for control messages must be done by the implementation of the `OnExecute` handler.

See also: TDaemon.OnStart (167), TDaemon.OnStop (168)

9.10.11 TDaemon.BeforeInstall

Synopsis: Called before the daemon will be installed

Declaration: `Property BeforeInstall : TDaemonEvent`

Visibility: published

Access: Read,Write

Description: `BeforeInstall` is called before the daemon is installed. It can be done to specify extra dependencies, or change the daemon description etc.

See also: TDaemon.AfterInstall (170), TDaemon.BeforeUnInstall (170), TDaemon.AfterUnInstall (170)

9.10.12 TDaemon.AfterInstall

Synopsis: Called after the daemon was installed

Declaration: `Property AfterInstall : TDaemonEvent`

Visibility: published

Access: Read,Write

Description: `AfterInstall` is called after the daemon was succesfully installed.

See also: TDaemon.BeforeInstall (169), TDaemon.BeforeUnInstall (170), TDaemon.AfterUnInstall (170)

9.10.13 TDaemon.BeforeUnInstall

Synopsis: Called before the daemon is uninstalled

Declaration: `Property BeforeUnInstall : TDaemonEvent`

Visibility: published

Access: Read,Write

Description: `BeforeUnInstall` is called before the daemon is uninstalled.

See also: TDaemon.BeforeInstall (169), TDaemon.AfterInstall (170), TDaemon.AfterUnInstall (170)

9.10.14 TDaemon.AfterUnInstall

Synopsis: Called after the daemon is uninstalled

Declaration: `Property AfterUnInstall : TDaemonEvent`

Visibility: published

Access: Read,Write

Description: `AfterUnInstall` is called after the daemon is succesfully uninstalled.

See also: TDaemon.BeforeInstall (169), TDaemon.AfterInstall (170), TDaemon.BeforeUnInstall (170)

9.10.15 TDaemon.OnControlCode

Synopsis: Called when a control code is received for the daemon

Declaration: `Property OnControlCode : TCustomControlCodeEvent`

Visibility: published

Access: Read,Write

Description: `OnControlCode` is called when the daemon receives a control code. If the daemon has not handled the control code, it should set the `Handled` parameter to `False`. By default it is set to `True`.

See also: daemonapp.architecture (151)

9.11 TDaemonApplication

9.11.1 Description

`TDaemonApplication` is the default TCustomDaemonApplication (160) descendent that is used to run the daemon application. It is possible to register an alternative `TCustomDaemonApplication` class (using RegisterDaemonApplicationClass (156)) to run the application in a different manner.

9.12 TDaemonController

9.12.1 Description

`TDaemonController` is a class that is used by the TDaemonApplication (171) class to control the daemon during runtime. The `TDaemonApplication` class instantiates an instance of `TDaemonController` for each daemon in the application and communicates with the daemon through the `TDaemonController` instance. It should rarely be necessary to access or use this class.

9.12.2 Method overview

Page	Property	Description
172	Controller	Controller
171	Create	Create a new instance of the `TDaemonController` class
172	Destroy	Free a `TDaemonController` instance.
172	Main	Daemon main entry point
173	ReportStatus	Report the status to the operating system.
172	StartService	Start the service

9.12.3 Property overview

Page	Property	Access	Description
174	CheckPoint		Send checkpoint signal to the operating system
173	Daemon	r	Daemon instance this controller controls.
173	LastStatus	r	Last reported status
173	Params	r	Parameters passed to the daemon

9.12.4 TDaemonController.Create

Synopsis: Create a new instance of the `TDaemonController` class

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` creates a new instance of the `TDaemonController` class. It should never be necessary to create a new instance manually, because the controllers are created by the global TDaemonApplication (171) instance, and `AOwner` will be set to the global TDaemonApplication (171) instance.

See also: TDaemonApplication (171), TDaemonController.Destroy (172)

9.12.5 TDaemonController.Destroy

Synopsis: Free a `TDaemonController` instance.

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` deallocates some resources allocated when the instance was created.

See also: TDaemonController.Create (171)

9.12.6 TDaemonController.StartService

Synopsis: Start the service

Declaration:
```
procedure StartService;  Virtual
```

Visibility: public

Description: `StartService` starts the service controlled by this instance.

Errors: None.

See also: TDaemonController.Main (172)

9.12.7 TDaemonController.Main

Synopsis: Daemon main entry point

Declaration:
```
procedure Main(Argc: DWord;Args: PPChar);  Virtual
```

Visibility: public

Description: `Main` is the service's main entry point, called when the system wants to start the service. The global application will call this function whenever required, with the appropriate arguments.

The standard implementation starts the daemon thread, and waits for it to stop. All other daemon action - such as responding to control code events - is handled by the thread.

Errors: If the daemon thread cannot be created, an exception is raised.

See also: TDaemonThread (180)

9.12.8 TDaemonController.Controller

Synopsis: Controller

Declaration:
```
procedure Controller(ControlCode: DWord;EventType: DWord;
                     EventData: Pointer);  Virtual
```

Visibility: public

Description: `Controller` is responsible for sending the control code to the daemon thread so it can be processed.

This routine is currently only used on windows, as there is no service manager on linux. Later on this may be changed to respond to signals on linux as well.

See also: TDaemon.OnControlCode (170)

9.12.9 TDaemonController.ReportStatus

Synopsis: Report the status to the operating system.

Declaration: `function ReportStatus : Boolean;  Virtual`

Visibility: public

Description: `ReportStatus` reports the status of the daemon to the operating system. On windows, this sends the current service status to the service manager. On other operating systems, this sends a message to the system log.

Errors: If an error occurs, an error message is sent to the system log.

See also: TDaemon.ReportStatus (166), TDaemonController.LastStatus (173)

9.12.10 TDaemonController.Daemon

Synopsis: Daemon instance this controller controls.

Declaration: `Property Daemon : TCustomDaemon`

Visibility: public

Access: Read

Description: `Daemon` is the daemon instance that is controller by this instance of the `TDaemonController` class.

9.12.11 TDaemonController.Params

Synopsis: Parameters passed to the daemon

Declaration: `Property Params : TStrings`

Visibility: public

Access: Read

Description: `Params` contains the parameters passed to the daemon application by the operating system, comparable to the application's command-line parameters. The property is set by the Main (172) method.

9.12.12 TDaemonController.LastStatus

Synopsis: Last reported status

Declaration: `Property LastStatus : TCurrentStatus`

Visibility: public

Access: Read

Description: `LastStatus` is the last status reported to the operating system.

See also: TDaemonController.ReportStatus (173)

9.12.13 TDaemonController.CheckPoint

Synopsis: Send checkpoint signal to the operating system

Declaration: `Property CheckPoint : DWord`

Visibility: public

Access:

Description: `CheckPoint` can be used to send a checkpoint signal during lengthy operations, to signal that a lengthy operation is in progress. This should be used mainly on windows, to signal the service manager that the service is alive.

See also: TDaemonController.ReportStatus (173)

9.13 TDaemonDef

9.13.1 Description

`TDaemonDef` contains the definition of a daemon in the application: The name of the daemon, which TCustomDaemon (157) descendent should be started to run the daemon, a description, and various other options should be set in this class. The global `TDaemonApplication` instance maintains a collection of `TDaemonDef` instances and will use these definitions to install or start the various daemons.

9.13.2 Method overview

Page	Property	Description
174	Create	Create a new `TDaemonDef` instance
175	Destroy	Free a `TDaemonDef` from memory

9.13.3 Property overview

Page	Property	Access	Description
175	DaemonClass	r	`TDaemon` class to use for this daemon
175	DaemonClassName	rw	Name of the `TDaemon` class to use for this daemon
176	Description	rw	Description of the daemon
176	DisplayName	rw	Displayed name of the daemon (service)
177	Enabled	rw	Is the daemon enabled or not
175	Instance	rw	Instance of the daemon class
178	LogStatusReport	rw	Log the status report to the system log
176	Name	rw	Name of the daemon (service)
177	OnCreateInstance	rw	Event called when a daemon in instantiated
177	Options	rw	Service options
176	RunArguments	rw	Additional command-line arguments when running daemon.
177	WinBindings	rw	Windows-specific bindings (windows only)

9.13.4 TDaemonDef.Create

Synopsis: Create a new `TDaemonDef` instance

Declaration: `constructor Create(ACollection: TCollection); Override`

Visibility: public

Description: `Create` initializes a new `TDaemonDef` instance. It should not be necessary to instantiate a definition manually, it is handled by the collection.

See also: TDaemonDefs (178)

9.13.5 TDaemonDef.Destroy

Synopsis: Free a `TDaemonDef` from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` removes the `TDaemonDef` from memory.

9.13.6 TDaemonDef.DaemonClass

Synopsis: `TDaemon` class to use for this daemon

Declaration: `Property DaemonClass : TCustomDaemonClass`

Visibility: public

Access: Read

Description: `DaemonClass` is the `TDaemon` class that is used when this service is requested. It is looked up in the application's global daemon mapper by it's name in DaemonClassName (175).

See also: TDaemonDef.DaemonClassName (175), TDaemonMapper (180)

9.13.7 TDaemonDef.Instance

Synopsis: Instance of the daemon class

Declaration: `Property Instance : TCustomDaemon`

Visibility: public

Access: Read,Write

Description: `Instance` points to the TDaemon (166) instance that is used when the service is in operation at runtime.

See also: TDaemonDef.DaemonClass (175)

9.13.8 TDaemonDef.DaemonClassName

Synopsis: Name of the `TDaemon` class to use for this daemon

Declaration: `Property DaemonClassName : String`

Visibility: published

Access: Read,Write

Description: `DaemonClassName` is the name of the `TDaemon` class that will be used whenever the service is needed. The name is used to look up the class pointer registered in the daemon mapper, when TCustomDaemonApplication.CreateDaemonInstance (160) creates an instance of the daemon.

See also: TDaemonDef.Instance (175), TDaemonDef.DaemonClass (175), RegisterDaemonClass (157)

9.13.9 TDaemonDef.Name

Synopsis: Name of the daemon (service)

Declaration: `Property Name : String`

Visibility: published

Access: Read,Write

Description: `Name` is the internal name of the daemon as it is known to the operating system.

See also: TDaemonDef.DisplayName (176)

9.13.10 TDaemonDef.Description

Synopsis: Description of the daemon

Declaration: `Property Description : String`

Visibility: published

Access: Read,Write

Description: `Description` is the description shown in the Windows service manager when managing this service. It is supplied to the windows service manager when the daemon is installed.

9.13.11 TDaemonDef.DisplayName

Synopsis: Displayed name of the daemon (service)

Declaration: `Property DisplayName : String`

Visibility: published

Access: Read,Write

Description: `DisplayName` is the displayed name of the daemon as it is known to the operating system.

See also: TDaemonDef.Name (176)

9.13.12 TDaemonDef.RunArguments

Synopsis: Additional command-line arguments when running daemon.

Declaration: `Property RunArguments : String`

Visibility: published

Access: Read,Write

Description: `RunArguments` specifies any additional command-line arguments that should be specified when running the daemon: these arguments will be passed to the service manager when registering the service on windows.

9.13.13 TDaemonDef.Options

Synopsis: Service options

Declaration: `Property Options : TDaemonOptions`

Visibility: published

Access: Read,Write

Description: `Options` tells the operating system which operations can be performed on the daemon while it is running.

This option is only used during the installation of the daemon.

9.13.14 TDaemonDef.Enabled

Synopsis: Is the daemon enabled or not

Declaration: `Property Enabled : Boolean`

Visibility: published

Access: Read,Write

Description: `Enabled` specifies whether a daemon should be installed, run or uninstalled. Disabled daemons are not installed, run or uninstalled.

9.13.15 TDaemonDef.WinBindings

Synopsis: Windows-specific bindings (windows only)

Declaration: `Property WinBindings : TWinBindings`

Visibility: published

Access: Read,Write

Description: `WinBindings` is used to group together the windows-specific properties of the daemon. This property is totally ignored on other platforms.

See also: TWinBindings (185)

9.13.16 TDaemonDef.OnCreateInstance

Synopsis: Event called when a daemon in instantiated

Declaration: `Property OnCreateInstance : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnCreateInstance` is called whenever an instance of the daemon is created. This can be used for instance when a single `TDaemon` class is used to run several services, to correctly initialize the TDaemon.

9.13.17 TDaemonDef.LogStatusReport

Synopsis: Log the status report to the system log

Declaration: `Property LogStatusReport : Boolean`

Visibility: published

Access: Read,Write

Description: `LogStatusReport` can be set to `True` to send the status reports also to the system log. This can be used to track the progress of the daemon.

See also: TDaemon.ReportStatus (166)

9.14 TDaemonDefs

9.14.1 Description

`TDaemonDefs` is the class of the global list of daemon definitions. It contains an item for each daemon in the application.

Normally it is not necessary to create an instance of `TDaemonDefs` manually. The global TCustomDaemonMapper (163) instance will create a collection and maintain it.

9.14.2 Method overview

Page	Property	Description
178	Create	Create a new instance of a `TDaemonDefs` collection.
179	DaemonDefByName	Find and return instance of daemon definition with given name.
179	FindDaemonDef	Find and return instance of daemon definition with given name.
179	IndexOfDaemonDef	Return index of daemon definition

9.14.3 Property overview

Page	Property	Access	Description
179	Daemons	rw	Indexed access to `TDaemonDef` instances

9.14.4 TDaemonDefs.Create

Synopsis: Create a new instance of a `TDaemonDefs` collection.

Declaration: `constructor Create(AOwner: TPersistent;AClass: TCollectionItemClass)`

Visibility: public

Description: `Create` creates a new instance of the `TDaemonDefs` collection. It keeps the `AOwner` parameter for future reference and calls the inherited constructor.

Normally it is not necessary to create an instance of `TDaemonDefs` manually. The global TCustomDaemonMapper (163) instance will create a collection and maintain it.

See also: TDaemonDef (174)

9.14.5 TDaemonDefs.IndexOfDaemonDef

Synopsis: Return index of daemon definition

Declaration: `function IndexOfDaemonDef(const DaemonName: String) : Integer`

Visibility: public

Description: `IndexOfDaemonDef` searches the collection for a TDaemonDef (174) instance with a name equal to `DaemonName`, and returns it's index. It returns -1 if no definition was found with this name. The search is case insensitive.

See also: TDaemonDefs.FindDaemonDef (179), TDaemonDefs.DaemonDefByName (179)

9.14.6 TDaemonDefs.FindDaemonDef

Synopsis: Find and return instance of daemon definition with given name.

Declaration: `function FindDaemonDef(const DaemonName: String) : TDaemonDef`

Visibility: public

Description: `FindDaemonDef` searches the list of daemon definitions and returns the TDaemonDef (174) instance whose name matches `DaemonName`. If no definition is found, `Nil` is returned.

See also: TDaemonDefs.IndexOfDaemonDef (179), TDaemonDefs.DaemonDefByName (179)

9.14.7 TDaemonDefs.DaemonDefByName

Synopsis: Find and return instance of daemon definition with given name.

Declaration: `function DaemonDefByName(const DaemonName: String) : TDaemonDef`

Visibility: public

Description: `FindDaemonDef` searches the list of daemon definitions and returns the TDaemonDef (174) instance whose name matches `DaemonName`. If no definition is found, an EDaemon (157) exception is raised.

The FindDaemonDef (179) call does not raise an error, but returns `Nil` instead.

Errors: If no definition is found, an EDaemon (157) exception is raised.

See also: TDaemonDefs.IndexOfDaemonDef (179), TDaemonDefs.FindDaemonDef (179)

9.14.8 TDaemonDefs.Daemons

Synopsis: Indexed access to `TDaemonDef` instances

Declaration: `Property Daemons[Index: Integer]: TDaemonDef; default`

Visibility: public

Access: Read,Write

Description: `Daemons` is the default property of `TDaemonDefs`, it gives access to the `TDaemonDef` instances in the collection.

See also: TDaemonDef (174)

9.15 TDaemonMapper

9.15.1 Description

TDaemonMapper is a direct descendent of TCustomDaemonMapper (163), but introduces no new functionality. It's sole purpose is to make it possible for an IDE to stream the TDaemonMapper instance.

For this purpose, it overrides the Create constructor and tries to find a resource with the same name as the class name, and tries to stream the instance from this resource.

If the instance should not be streamed, the CreateNew (180) constructor can be used instead.

9.15.2 Method overview

Page	Property	Description
180	Create	Create a new TDaemonMapper instance and initializes it from streamed resources.
180	CreateNew	Create a new TDaemonMapper instance without initialization

9.15.3 TDaemonMapper.Create

Synopsis: Create a new TDaemonMapper instance and initializes it from streamed resources.

Declaration:
```
constructor Create(AOwner: TComponent);  Override
```

Visibility: default

Description: Create initializes a new instance of TDaemonMapper and attempts to read the component from resources compiled in the application.

If the instance should not be streamed, the CreateNew (180) constructor can be used instead.

Errors: If no streaming system is found, or no resource exists for the class, an exception is raised.

See also: TDaemonMapper.CreateNew (180)

9.15.4 TDaemonMapper.CreateNew

Synopsis: Create a new TDaemonMapper instance without initialization

Declaration:
```
constructor CreateNew(AOwner: TComponent;Dummy: Integer)
```

Visibility: default

Description: CreateNew itializes a new instance of TDaemonMapper. In difference with the Create constructor, it does not attempt to read the component from a stream.

See also: TDaemonMapper.Create (180)

9.16 TDaemonThread

9.16.1 Description

TDaemonThread is the thread in which the daemons in the application are run. Each daemon is run in it's own thread.

It should not be necessary to create these threads manually, the TDaemonController (171) class will take care of this.

9.16.2 Method overview

Page	Property	Description
181	CheckControlMessage	Check if a control message has arrived
182	ContinueDaemon	Continue the daemon
181	Create	Create a new thread
181	Execute	Run the daemon
183	InterrogateDaemon	Report the daemon status
182	PauseDaemon	Pause the daemon
182	ShutDownDaemon	Shut down daemon
182	StopDaemon	Stops the daemon

9.16.3 Property overview

Page	Property	Access	Description
183	Daemon	r	Daemon instance

9.16.4 TDaemonThread.Create

Synopsis: Create a new thread

Declaration: `constructor Create(ADaemon: TCustomDaemon)`

Visibility: public

Description: `Create` creates a new thread instance. It initializes the `Daemon` property with the passed `ADaemon`. The thread is created suspended.

See also: TDaemonThread.Daemon (183)

9.16.5 TDaemonThread.Execute

Synopsis: Run the daemon

Declaration: `procedure Execute;  Override`

Visibility: public

Description: `Execute` starts executing the daemon and waits till the daemon stops. It also listens for control codes for the daemon.

See also: TDaemon.Execute (166)

9.16.6 TDaemonThread.CheckControlMessage

Synopsis: Check if a control message has arrived

Declaration: `procedure CheckControlMessage(WaitForMessage: Boolean)`

Visibility: public

Description: `CheckControlMessage` checks if a control message has arrived for the daemon and executes the appropriate daemon message. If the parameter `WaitForMessage` is `True`, then the routine waits for the message to arrive. If it is `False` and no message is present, it returns at once.

9.16.7 TDaemonThread.StopDaemon

Synopsis: Stops the daemon

Declaration: `function StopDaemon : Boolean;  Virtual`

Visibility: public

Description: `StopDaemon` attempts to stop the daemon using its TDaemon.Stop (166) method, and terminates the thread.

See also: TDaemon.Stop (166), TDaemonThread.PauseDaemon (182), TDaemonThread.ShutDownDaemon (182)

9.16.8 TDaemonThread.PauseDaemon

Synopsis: Pause the daemon

Declaration: `function PauseDaemon : Boolean;  Virtual`

Visibility: public

Description: `PauseDaemon` attempts to stop the daemon using its TDaemon.Pause (166) method, and suspends the thread. It returns `True` if the attempt was succesful.

See also: TDaemon.Pause (166), TDaemonThread.StopDaemon (182), TDaemonThread.ContinueDaemon (182), TDaemonThread.ShutDownDaemon (182)

9.16.9 TDaemonThread.ContinueDaemon

Synopsis: Continue the daemon

Declaration: `function ContinueDaemon : Boolean;  Virtual`

Visibility: public

Description: `ContinueDaemon` attempts to stop the daemon using its TDaemon.Continue (166) method. It returns `True` if the attempt was succesful.

See also: TDaemon.Continue (166), TDaemonThread.StopDaemon (182), TDaemonThread.PauseDaemon (182), TDaemonThread.ShutDownDaemon (182)

9.16.10 TDaemonThread.ShutDownDaemon

Synopsis: Shut down daemon

Declaration: `function ShutDownDaemon : Boolean;  Virtual`

Visibility: public

Description: `ShutDownDaemon` shuts down the daemon. This happens normally only when the system is shut down and the daemon didn't respond to the stop request. The return result is the result of the TDaemon.Shutdown (166) function. The thread is terminated by this method.

See also: TDaemon.Shutdown (166), TDaemonThread.StopDaemon (182), TDaemonThread.PauseDaemon (182), TDaemonThread.ContinueDaemon (182)

9.16.11 TDaemonThread.InterrogateDaemon

Synopsis: Report the daemon status

Declaration: `function InterrogateDaemon : Boolean;  Virtual`

Visibility: public

Description: `InterrogateDaemon` simply calls TDaemon.ReportStatus (166) for the daemon that is running in this thread. It always returns `True`.

See also: TDaemon.ReportStatus (166)

9.16.12 TDaemonThread.Daemon

Synopsis: Daemon instance

Declaration: `Property Daemon : TCustomDaemon`

Visibility: public

Access: Read

Description: `Daemon` is the daemon instance which is running in this thread.

See also: TDaemon (166)

9.17 TDependencies

9.17.1 Description

`TDependencies` is just a descendent of `TCollection` which contains a series of dependencies on other services. It overrides the default property of `TCollection` to return TDependency (184) instances.

9.17.2 Method overview

Page	Property	Description
183	Create	Create a new instance of a `TDependencies` collection.

9.17.3 Property overview

Page	Property	Access	Description
184	Items	rw	Default property override

9.17.4 TDependencies.Create

Synopsis: Create a new instance of a `TDependencies` collection.

Declaration: `constructor Create(AOwner: TPersistent)`

Visibility: public

Description: `Create` Create a new instance of a `TDependencies` collection.

9.17.5 TDependencies.Items

Synopsis: Default property override

Declaration: `Property Items[Index: Integer]: TDependency; default`

Visibility: public

Access: Read,Write

Description: `Items` overrides the default property of `TCollection` so the items are of type TDependency (184).

See also: TDependency (184)

9.18 TDependency

9.18.1 Description

`TDependency` is a collection item used to specify dependencies on other daemons (services) in windows. It is used only on windows and when installing the daemon: changing the dependencies of a running daemon has no effect.

9.18.2 Method overview

Page	Property	Description
184	Assign	Assign `TDependency` instance to another

9.18.3 Property overview

Page	Property	Access	Description
185	IsGroup	rw	Name refers to a service group
184	Name	rw	Name of the service

9.18.4 TDependency.Assign

Synopsis: Assign `TDependency` instance to another

Declaration: `procedure Assign(Source: TPersistent);  Override`

Visibility: public

Description: `Assign` is overridden by `TDependency` to copy all properties from one instance to another.

9.18.5 TDependency.Name

Synopsis: Name of the service

Declaration: `Property Name : String`

Visibility: published

Access: Read,Write

Description: `Name` is the name of a service or service group that the current daemon depends on.

See also: TDependency.IsGroup (185)

9.18.6 TDependency.IsGroup

Synopsis: Name refers to a service group

Declaration: `Property IsGroup : Boolean`

Visibility: published

Access: Read,Write

Description: `IsGroup` can be set to `True` to indicate that `Name` refers to the name of a service group.

See also: TDependency.Name (184)

9.19 TWinBindings

9.19.1 Description

`TWinBindings` contains windows-specific properties for the daemon definition (in TDaemonDef.WinBindings (177)). If the daemon should not run on Windows, then the properties can be ignored.

9.19.2 Method overview

Page	Property	Description
186	Assign	Copies all properties
185	Create	Create a new `TWinBindings` instance
186	Destroy	Remove a `TWinBindings` instance from memory

9.19.3 Property overview

Page	Property	Access	Description
187	Dependencies	rw	Service dependencies
186	ErrCode	rw	Service specific error code
189	ErrorSeverity	rw	Error severity in case of startup failure
187	GroupName	rw	Service group name
188	IDTag	rw	Location in the service group
187	Password	rw	Password for service startup
189	ServiceType	rw	Type of service
188	StartType	rw	Service startup type.
187	UserName	rw	Username to run service as
188	WaitHint	rw	Timeout wait hint
186	Win32ErrCode	rw	General windows error code

9.19.4 TWinBindings.Create

Synopsis: Create a new `TWinBindings` instance

Declaration: `constructor Create`

Visibility: public

Description: `Create` initializes various properties such as the dependencies.

See also: TDaemonDef (174), TDaemonDef.WinBindings (177), TWinBindings.Dependencies (187)

9.19.5 TWinBindings.Destroy

Synopsis: Remove a `TWinBindings` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` cleans up the `TWinBindings` instance.

See also: TWinBindings.Dependencies (187), TWinBindings.Create (185)

9.19.6 TWinBindings.Assign

Synopsis: Copies all properties

Declaration: `procedure Assign(Source: TPersistent); Override`

Visibility: public

Description: `Assign` is overridden by `TWinBindings` so all properties are copied from `Source` to the `TWinBindings` instance.

9.19.7 TWinBindings.ErrCode

Synopsis: Service specific error code

Declaration: `Property ErrCode : DWord`

Visibility: public

Access: Read,Write

Description: `ErrCode` contains a service specific error code that is reported with TDaemon.ReportStatus (166) to the windows service manager. If it is zero, then the contents of Win32ErrCode (186) are reported. If it is nonzero, then the windows-errorcode is set to `ERROR_SERVICE_SPECIFIC_ERROR`.

See also: TWinBindings.Win32ErrCode (186)

9.19.8 TWinBindings.Win32ErrCode

Synopsis: General windows error code

Declaration: `Property Win32ErrCode : DWord`

Visibility: public

Access: Read,Write

Description: `Win32ErrCode` is a general windows service error code that can be reported with TDaemon.ReportStatus (166) to the windows service manager. It is sent if ErrCode (186) is zero.

See also: TWinBindings.ErrCode (186)

9.19.9 TWinBindings.Dependencies

Synopsis: Service dependencies

Declaration: `Property Dependencies : TDependencies`

Visibility: published

Access: Read,Write

Description: `Dependencies` contains the list of other services (or service groups) that this service depends on. Windows will first attempt to start these services prior to starting this service. If they cannot be started, then the service will not be started either.

This property is only used during installation of the service.

9.19.10 TWinBindings.GroupName

Synopsis: Service group name

Declaration: `Property GroupName : String`

Visibility: published

Access: Read,Write

Description: `GroupName` specifies the name of a service group that the service belongs to. If it is empty, then the service does not belong to any group.

This property is only used during installation of the service.

See also: TDependency.IsGroup (185)

9.19.11 TWinBindings.Password

Synopsis: Password for service startup

Declaration: `Property Password : String`

Visibility: published

Access: Read,Write

Description: `Password` contains the service password: if the service is started with credentials other than one of the system users, then the password for the user must be entered here.

This property is only used during installation of the service.

See also: TWinBindings.UserName (187)

9.19.12 TWinBindings.UserName

Synopsis: Username to run service as

Declaration: `Property UserName : String`

Visibility: published

Access: Read,Write

Description: `Username` specifies the name of a user whose credentials should be used to run the service. If it is left empty, the service is run as the system user. The password can be set in the Password (187) property.

This property is only used during installation of the service.

See also: TWinBindings.Password (187)

9.19.13 TWinBindings.StartType

Synopsis: Service startup type.

Declaration: `Property StartType : TStartType`

Visibility: published

Access: Read,Write

Description: `StartType` specifies when the service should be started during system startup.

This property is only used during installation of the service.

9.19.14 TWinBindings.WaitHint

Synopsis: Timeout wait hint

Declaration: `Property WaitHint : Integer`

Visibility: published

Access: Read,Write

Description: `WaitHint` specifies the estimated time for a start/stop/pause or continue operation (in milliseconds). Reportstatus should be called prior to this time to report the next status.

See also: TDaemon.ReportStatus (166)

9.19.15 TWinBindings.IDTag

Synopsis: Location in the service group

Declaration: `Property IDTag : DWord`

Visibility: published

Access: Read,Write

Description: `IDTag` contains the location of the service in the service group after installation of the service. It should not be set, it is reported by the service manager.

This property is only used during installation of the service.

9.19.16 TWinBindings.ServiceType

Synopsis: Type of service

Declaration: `Property ServiceType : TServiceType`

Visibility: published

Access: Read,Write

Description: `ServiceType` specifies what kind of service is being installed.

This property is only used during installation of the service.

9.19.17 TWinBindings.ErrorSeverity

Synopsis: Error severity in case of startup failure

Declaration: `Property ErrorSeverity : TErrorSeverity`

Visibility: published

Access: Read,Write

Description: `ErrorSeverity` can be used at installation time to tell the windows service manager how to behave when the service fails to start during system startup.

This property is only used during installation of the service.

Chapter 10

Reference for unit 'dbugintf'

10.1 Writing a debug server

Writing a debug server is relatively easy. It should instantiate a `TSimpleIPCServer` class from the SimpleIPC (190) unit, and use the `DebugServerID` as `ServerID` identification. This constant, as well as the record containing the message which is sent between client and server is defined in the msgintf unit.

The dbugintf unit relies on the SimpleIPC (190) mechanism to communicate with the debug server, hence it works on all platforms that have a functional version of that unit. It also uses `TProcess` to start the debug server if needed, so the process (190) unit should also be functional.

10.2 Overview

Use dbugintf to add debug messages to your application. The messages are not sent to standard output, but are sent to a debug server process which collects messages from various clients and displays them somehow on screen.

The unit is transparant in its use: it does not need initialization, it will start the debug server by itself if it can find it: the program should be called debugserver and should be in the `PATH`. When the first debug message is sent, the unit will initialize itself.

The FCL contains a sample debug server (dbugsvr) which can be started in advance, and which writes debug message to the console (both on Windows and Linux). The Lazarus project contains a visual application which displays the messages in a GUI.

The dbugintf unit relies on the SimpleIPC (190) mechanism to communicate with the debug server, hence it works on all platforms that have a functional version of that unit. It also uses `TProcess` to start the debug server if needed, so the process (190) unit should also be functional.

10.3 Constants, types and variables

10.3.1 Resource strings

```
SEntering = '> Entering '
```

String used when sending method enter message.

```
SExiting = '< Exiting '
```

String used when sending method exit message.

```
SProcessID = 'Process %s'
```

String used when sending identification message to the server.

```
SSeparator = '>-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-<'
```

String used when sending a separator line.

```
SServerStartFailed = 'Failed to start debugserver. (%s)'
```

String used to display an error message when the start of the debug server failed

10.3.2 Constants

```
SendError : String = ''
```

Whenever a call encounteres an exception, the exception message is stored in this variable.

10.3.3 Types

```
TDebugLevel = (dlInformation,dlWarning,dlError)
```

Table 10.1: Enumeration values for type TDebugLevel

Value	Explanation
dlError	Error message
dlInformation	Informational message
dlWarning	Warning message

`TDebugLevel` indicates the severity level of the debug message to be sent. By default, an informational message is sent.

10.4 Procedures and functions

10.4.1 GetDebuggingEnabled

Synopsis: Check if sending of debug messages is enabled.

Declaration: `function GetDebuggingEnabled : Boolean`

Visibility: default

Description: `GetDebuggingEnabled` returns the value set by the last call to `SetDebuggingEnabled`. It is `True` by default.

See also: SetDebuggingEnabled (195), SendDebug (192)

10.4.2 InitDebugClient

Synopsis: Initialize the debug client.

Declaration: `function InitDebugClient : Boolean`

Visibility: default

Description: `InitDebugClient` starts the debug server and then performs all necessary initialization of the debug IPC communication channel.

Normally this function should not be called. The SendDebug (192) call will initialize the debug client when it is first called.

Errors: None.

See also: SendDebug (192), StartDebugServer (195)

10.4.3 SendBoolean

Synopsis: Send the value of a boolean variable

Declaration: `procedure SendBoolean(const Identifier: String;const Value: Boolean)`

Visibility: default

Description: `SendBoolean` is a simple wrapper around SendDebug (192) which sends the name and value of a boolean value as an informational message.

Errors: None.

See also: SendDebug (192), SendDateTime (192), SendInteger (194), SendPointer (195)

10.4.4 SendDateTime

Synopsis: Send the value of a `TDateTime` variable.

Declaration: `procedure SendDateTime(const Identifier: String;const Value: TDateTime)`

Visibility: default

Description: `SendDateTime` is a simple wrapper around SendDebug (192) which sends the name and value of an integer value as an informational message. The value is converted to a string using the DateTime-ToStr (**??**) call.

Errors: None.

See also: SendDebug (192), SendBoolean (192), SendInteger (194), SendPointer (195)

10.4.5 SendDebug

Synopsis: Send a message to the debug server.

Declaration: `procedure SendDebug(const Msg: String)`

Visibility: default

Description: `SendDebug` sends the message `Msg` to the debug server as an informational message (debug level `dlInformation`). If no debug server is running, then an attempt will be made to start the server first.

The binary that is started is called debugserver and should be somewhere on the `PATH`. A sample binary which writes received messages to standard output is included in the FCL, it is called dbugsrv. This binary can be renamed to debugserver or can be started before the program is started.

Errors: Errors are silently ignored, any exception messages are stored in SendError (191).

See also: SendDebugEx (193), SendDebugFmt (193), SendDebugFmtEx (193)

10.4.6 SendDebugEx

Synopsis: Send debug message other than informational messages

Declaration:
```
procedure SendDebugEx(const Msg: String;MType: TDebugLevel)
```

Visibility: default

Description: `SendDebugEx` allows to specify the debug level of the message to be sent in `MType`. By default, SendDebug (192) uses informational messages.

Other than that the function of `SendDebugEx` is equal to that of `SendDebug`

Errors: None.

See also: SendDebug (192), SendDebugFmt (193), SendDebugFmtEx (193)

10.4.7 SendDebugFmt

Synopsis: Format and send a debug message

Declaration:
```
procedure SendDebugFmt(const Msg: String;const Args: Array of const)
```

Visibility: default

Description: `SendDebugFmt` is a utility routine which formats a message by passing `Msg` and `Args` to Format (**??**) and sends the result to the debug server using SendDebug (192). It exists mainly to avoid the `Format` call in calling code.

Errors: None.

See also: SendDebug (192), SendDebugEx (193), SendDebugFmtEx (193), #rtl.sysutils.format (**??**)

10.4.8 SendDebugFmtEx

Synopsis: Format and send message with alternate type

Declaration:
```
procedure SendDebugFmtEx(const Msg: String;const Args: Array of const;
                         MType: TDebugLevel)
```

Visibility: default

Description: `SendDebugFmtEx` is a utility routine which formats a message by passing `Msg` and `Args` to Format (**??**) and sends the result to the debug server using SendDebugEx (193) with Debug level `MType`. It exists mainly to avoid the `Format` call in calling code.

Errors: None.

See also: SendDebug (192), SendDebugEx (193), SendDebugFmt (193), #rtl.sysutils.format (**??**)

10.4.9 SendInteger

Synopsis: Send the value of an integer variable.

Declaration:
```
procedure SendInteger(const Identifier: String;const Value: Integer;
                      HexNotation: Boolean)
```

Visibility: default

Description: `SendInteger` is a simple wrapper around SendDebug (192) which sends the name and value of an integer value as an informational message. If `HexNotation` is `True`, then the value will be displayed using hexadecimal notation.

Errors: None.

See also: SendDebug (192), SendBoolean (192), SendDateTime (192), SendPointer (195)

10.4.10 SendMethodEnter

Synopsis: Send method enter message

Declaration:
```
procedure SendMethodEnter(const MethodName: String)
```

Visibility: default

Description: `SendMethodEnter` sends a "Entering `MethodName`" message to the debug server. After that it increases the message indentation (currently 2 characters). By sending a corresponding SendMethodExit (194), the indentation of messages can be decreased again.

By using the `SendMethodEnter` and `SendMethodExit` methods at the beginning and end of a procedure/method, it is possible to visually trace program execution.

Errors: None.

See also: SendDebug (192), SendMethodExit (194), SendSeparator (195)

10.4.11 SendMethodExit

Synopsis: Send method exit message

Declaration:
```
procedure SendMethodExit(const MethodName: String)
```

Visibility: default

Description: `SendMethodExit` sends a "Exiting `MethodName`" message to the debug server. After that it decreases the message indentation (currently 2 characters). By sending a corresponding SendMethodEnter (194), the indentation of messages can be increased again.

By using the `SendMethodEnter` and `SendMethodExit` methods at the beginning and end of a procedure/method, it is possible to visually trace program execution.

Note that the indentation level will not be made negative.

Errors: None.

See also: SendDebug (192), SendMethodEnter (194), SendSeparator (195)

10.4.12 SendPointer

Synopsis: Send the value of a pointer variable.

Declaration: `procedure SendPointer(const Identifier: String;const Value: Pointer)`

Visibility: default

Description: `SendInteger` is a simple wrapper around SendDebug (192) which sends the name and value of a pointer value as an informational message. The pointer value is displayed using hexadecimal notation.

Errors: None.

See also: SendDebug (192), SendBoolean (192), SendDateTime (192), SendInteger (194)

10.4.13 SendSeparator

Synopsis: Send a separator message

Declaration: `procedure SendSeparator`

Visibility: default

Description: `SendSeparator` is a simple wrapper around SendDebug (192) which sends a short horizontal line to the debug server. It can be used to visually separate execution of blocks of code or blocks of values.

Errors: None.

See also: SendDebug (192), SendMethodEnter (194), SendMethodExit (194)

10.4.14 SetDebuggingEnabled

Synopsis: Temporary enables or disables debugging

Declaration: `procedure SetDebuggingEnabled(const AValue: Boolean)`

Visibility: default

Description: `SetDebuggingEnabled` can be used to temporarily enable or disable sending of debug messages: this allows to control the amount of messages sent to the debug server without having to remove the SendDebug (192) statements. By default, debugging is enabled. If set to false, debug messages are simply discarded till debugging is enabled again.

A value of `True` enables sending of debug messages. A value of `False` disables sending.

Errors: None.

See also: GetDebuggingEnabled (191), SendDebug (192)

10.4.15 StartDebugServer

Synopsis: Start the debug server

Declaration: `function StartDebugServer : Integer`

Visibility: default

Description: `StartDebugServer` attempts to start the debug server. The process started is called debugserver and should be located in the `PATH`.

Normally this function should not be called. The SendDebug (192) call will attempt to start the server by itself if it is not yet running.

Errors: On error, `False` is returned.

See also: SendDebug (192), InitDebugClient (192)

Chapter 11

Reference for unit 'dbugmsg'

11.1 Used units

Table 11.1: Used units by unit 'dbugmsg'

Name	Page
Classes	??

11.2 Overview

dbugmsg is an auxialiary unit used in the dbugintf (190) unit. It defines the message protocol used between the debug unit and the debug server.

11.3 Constants, types and variables

11.3.1 Constants

```
DebugServerID : String = 'fpcdebugserver'
```

`DebugServerID` is a string which is used when creating the message protocol, it is used when identifying the server in the (platform dependent) client-server protocol.

```
lctError = 2
```

`lctError` is the identification of error messages.

```
lctIdentify = 3
```

`lctIdentify` is sent by the client to a server when it first connects. It's the first message, and contains the name of client application.

```
lctInformation = 0
```

lctInformation is the identification of informational messages.

lctStop = -1

lctStop is sent by the client to a server when it disconnects.

lctWarning = 1

lctWarning is the identification of warning messages.

11.3.2 Types

```
TDebugMessage = record
  MsgType : Integer;
  MsgTimeStamp : TDateTime;
  Msg : String;
end
```

TDebugMessage is a record that describes the message passed from the client to the server. It should not be passed directly in shared memory, as the string containing the message is allocated on the heap. Instead, the WriteDebugMessageToStream (199) and ReadDebugMessageFromStream (198) can be used to read or write the message from/to a stream.

11.4 Procedures and functions

11.4.1 DebugMessageName

Synopsis: Return the name of the debug message

Declaration: `function DebugMessageName(msgType: Integer) : String`

Visibility: default

Description: DebugMessageName returns the name of the message type. It can be used to examine the MsgType field of a TDebugMessage (198) record, and if msgType contains a known type, it returns a string describing this type.

Errors: If MsgType contains an unknown type, 'Unknown' is returned.

11.4.2 ReadDebugMessageFromStream

Synopsis: Read a message from stream

Declaration:
```
procedure ReadDebugMessageFromStream(AStream: TStream;
                                     var Msg: TDebugMessage)
```

Visibility: default

Description: ReadDebugMessageFromStream reads a TDebugMessage (198) record (Msg) from the stream AStream.

The record is not read in a byte-ordering safe way, i.e. it cannot be exchanged between little- and big-endian systems.

Errors: If the stream contains not enough bytes or is malformed, then an exception may be raised.

See also: TDebugMessage (198), WriteDebugMessageToStream (199)

11.4.3 WriteDebugMessageToStream

Synopsis: Write a message to stream

Declaration:
```
procedure WriteDebugMessageToStream(AStream: TStream;
                                    const Msg: TDebugMessage)
```

Visibility: default

Description: `WriteDebugMessageFromStream` writes a TDebugMessage (198) record (`Msg`) to the stream `AStream`.

The record is not written in a byte-ordering safe way, i.e. it cannot be exchanged between little- and big-endian systems.

Errors: A stream write error may occur if the stream cannot be written to.

See also: TDebugMessage (198), ReadDebugMessageToStream (197)

Chapter 12

Reference for unit 'eventlog'

12.1 Used units

Table 12.1: Used units by unit 'eventlog'

Name	Page
Classes	??
sysutils	??

12.2 Overview

The EventLog unit implements the TEventLog (202) component, which is a component that can be used to send log messages to the system log (if it is available) or to a file.

12.3 Constants, types and variables

12.3.1 Resource strings

```
SErrLogFailedMsg = 'Failed to log entry (Error: %s)'
```

Message used to format an error when an error exception is raised.

```
SLogCustom = 'Custom (%d)'
```

Custom message formatting string

```
SLogDebug = 'Debug'
```

Debug message name

```
SLogError = 'Error'
```

Error message name

```
SLogInfo = 'Info'
```

Informational message name

```
SLogWarning = 'Warning'
```

Warning message name

12.3.2 Types

```
TEventType = (etCustom,etInfo,etWarning,etError,etDebug)
```

Table 12.2: Enumeration values for type TEventType

Value	Explanation
etCustom	Custom event type.
etDebug	Debug event
etError	Error event
etInfo	Informational event
etWarning	Warning event

`TEventType` determines the type of event. Depending on the system logger, the log event may end up in different places, or may be displayed in a different manner. A suitable mapping is shown for each system. In the case of Windows, the formatting of the message is done differently, and a different icon is shown for each type of message.

```
TLogCategoryEvent = procedure(Sender: TObject;var Code: Word) of object
```

`TLogCategoryEvent` is the event type for the TEventLog.OnGetCustomCategory (208) event handler. It should return a OS event catagory code for the `etCustom` log event type in the `Code` parameter.

```
TLogCodeEvent = procedure(Sender: TObject;var Code: DWord) of object
```

`TLogCodeEvent` is the event type for the OnGetCustomEvent (209) and OnGetCustomEventID (208) event handlers. It should return a OS system log code for the `etCustom` log event or event ID type in the `Code` parameter.

```
TLogType = (ltSystem,ltFile)
```

Table 12.3: Enumeration values for type TLogType

Value	Explanation
ltFile	Write to file
ltSystem	Use the system log

`TLogType` determines where the log messages are written. It is the type of the TEventLog.LogType (206) property. It can have 2 values:

ltFile This is used to write all messages to file. if no system logging mechanism exists, this is used as a fallback mechanism.

ltSystem This is used to send all messages to the system log mechanism. Which log mechanism this is, depends on the operating system.

12.4 ELogError

12.4.1 Description

`ELogError` is the exception used in the TEventLog (202) component to indicate errors.

12.5 TEventLog

12.5.1 Description

`TEventLog` is a component which can be used to send messages to the system log. In case no system log exists (such as on Windows 95/98 or DOS), the messages are written to a file. Messages can be logged using the general Log (204) call, or the specialized Warning (204), Error (205), Info (205) or Debug (205) calls, which have the event type predefined.

12.5.2 Method overview

Page	Property	Description
205	Debug	Log a debug message
203	Destroy	Clean up `TEventLog` instance
205	Error	Log an error message to
203	EventTypeToString	Create a string representation of an event type
205	Info	Log an informational message
204	Log	Log a message to the system log.
203	RegisterMessageFile	Register message file
204	Warning	Log a warning message.

12.5.3 Property overview

Page	Property	Access	Description
206	Active	rw	Activate the log mechanism
207	CustomLogType	rw	Custom log type ID
206	DefaultEventType	rw	Default event type for the Log (204) call.
208	EventIDOffset	rw	Offset for event ID messages identifiers
207	FileName	rw	File name for log file
205	Identification	rw	Identification string for messages
206	LogType	rw	Log type
208	OnGetCustomCategory	rw	Event to retrieve custom message category
209	OnGetCustomEvent	rw	Event to retrieve custom event Code
208	OnGetCustomEventID	rw	Event to retrieve custom event ID
206	RaiseExceptionOnError	rw	Determines whether logging errors are reported or ignored
207	TimeStampFormat	rw	Format for the timestamp string

12.5.4 TEventLog.Destroy

Synopsis: Clean up `TEventLog` instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the `TEventLog` instance. It cleans any log structures that might have been set up to perform logging, by setting the Active (206) property to False.

See also: TEventLog.Active (206)

12.5.5 TEventLog.EventTypeToString

Synopsis: Create a string representation of an event type

Declaration: `function EventTypeToString(E: TEventType) : String`

Visibility: public

Description: `EventTypeToString` converts the event type `E` to a suitable string representation for logging purposes. It's mainly used when writing messages to file, as the system log usually has it's own mechanisms for displaying the various event types.

See also: TEventType (201)

12.5.6 TEventLog.RegisterMessageFile

Synopsis: Register message file

Declaration: `function RegisterMessageFile(AFileName: String) : Boolean;  Virtual`

Visibility: public

Description: `RegisterMessageFile` is used on Windows to register the file `AFileName` containing the formatting strings for the system messages. This should be a file containing resource strings. If `AFileName` is empty, the filename of the application binary is substituted.

When a message is logged to the windows system log, Windows looks for a formatting string in the file registered with this call.

There are 2 kinds of formatting strings:

Category strings these should be numbered from 1 to 4

1Should contain the description of the `etInfo` event type.

2Should contain the description of the `etWarning` event type.

4Should contain the description of the `etError` event type.

4Should contain the description of the `etDebug` event type.

None of these strings should have a string substitution placeholder.

The second type of strings are the **message definitions**. Their number starts at EventIDOffset (208) (default is 1000) and each string should have 1 placeholder.

Free Pascal comes with a fclel.res resource file which contains default values for the 8 strings, in english. It can be linked in the application binary with the statement

```
{$R fclel.res}
```

This file is generated from the fclel.mc and fclel.rc files that are distributed with the Free Pascal sources.

If the strings are not registered, windows will still display the event messages, but they will not be formatted nicely.

Note that while any messages logged with the event logger are displayed in the event viewern Windows locks the file registered here. This usually means that the binary is locked.

On non-windows operating systems, this call is ignored.

Errors: If `AFileName` is invalid, false is returned.

12.5.7 TEventLog.Log

Synopsis: Log a message to the system log.

Declaration:
```
procedure Log(EventType: TEventType;Msg: String);  Overload
procedure Log(EventType: TEventType;Fmt: String;Args: Array of const)
             ;  Overload
procedure Log(Msg: String);  Overload
procedure Log(Fmt: String;Args: Array of const);  Overload
```

Visibility: public

Description: `Log` sends a log message to the system log. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters. If `EventType` is specified, then it is used as the message event type. If `EventType` is omitted, then the event type is determined from DefaultEventType (206).

If `EventType` is `etCustom`, then the OnGetCustomEvent (209), OnGetCustomEventID (208) and OnGetCustomCategory (208).

The other logging calls: Info (205), Warning (204), Error (205) and Debug (205) use the `Log` call to do the actual work.

See also: TEventLog.Info (205), TEventLog.Warning (204), TEventLog.Error (205), TEventLog.Debug (205), TEventLog.OnGetCustomEvent (209), TEventLog.OnGetCustomEventID (208), TEventLog.OnGetCustomCategory (208)

12.5.8 TEventLog.Warning

Synopsis: Log a warning message.

Declaration:
```
procedure Warning(Msg: String);  Overload
procedure Warning(Fmt: String;Args: Array of const);  Overload
```

Visibility: public

Description: `Warning` is a utility function which logs a message with the `etWarning` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: TEventLog.Log (204), TEventLog.Info (205), TEventLog.Error (205), TEventLog.Debug (205)

12.5.9 TEventLog.Error

Synopsis: Log an error message to

Declaration:
```
procedure Error(Msg: String);  Overload
procedure Error(Fmt: String;Args: Array of const);  Overload
```

Visibility: public

Description: `Error` is a utility function which logs a message with the `etError` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: TEventLog.Log (204), TEventLog.Info (205), TEventLog.Warning (204), TEventLog.Debug (205)

12.5.10 TEventLog.Debug

Synopsis: Log a debug message

Declaration:
```
procedure Debug(Msg: String);  Overload
procedure Debug(Fmt: String;Args: Array of const);  Overload
```

Visibility: public

Description: `Debug` is a utility function which logs a message with the `etDebug` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: TEventLog.Log (204), TEventLog.Info (205), TEventLog.Warning (204), TEventLog.Error (205)

12.5.11 TEventLog.Info

Synopsis: Log an informational message

Declaration:
```
procedure Info(Msg: String);  Overload
procedure Info(Fmt: String;Args: Array of const);  Overload
```

Visibility: public

Description: `Info` is a utility function which logs a message with the `etInfo` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: TEventLog.Log (204), TEventLog.Warning (204), TEventLog.Error (205), TEventLog.Debug (205)

12.5.12 TEventLog.Identification

Synopsis: Identification string for messages

Declaration:
```
Property Identification : String
```

Visibility: published

Access: Read,Write

Description: `Identification` is used as a string identifying the source of the messages in the system log. If it is empty, the filename part of the application binary is used.

See also: TEventLog.Active (206), TEventLog.TimeStampFormat (207)

12.5.13 TEventLog.LogType

Synopsis: Log type

Declaration: `Property LogType : TLogType`

Visibility: published

Access: Read,Write

Description: `LogType` is the type of the log: if it is `ltSystem`, then the system log is used, if it is available. If it is `ltFile` or there is no system log available, then the log messages are written to a file. The name for the log file is taken from the FileName (207) property.

See also: TEventLog.FileName (207)

12.5.14 TEventLog.Active

Synopsis: Activate the log mechanism

Declaration: `Property Active : Boolean`

Visibility: published

Access: Read,Write

Description: `Active` determines whether the log mechanism is active: if set to `True`, the component connects to the system log mechanism, or opens the log file if needed. Any attempt to log a message while the log is not active will try to set this property to `True`. Disconnecting from the system log or closing the log file is done by setting the `Active` property to `False`.

If the connection to the system logger fails, or the log file cannot be opened, then setting this property may result in an exception.

See also: TEventLog.Log (204)

12.5.15 TEventLog.RaiseExceptionOnError

Synopsis: Determines whether logging errors are reported or ignored

Declaration: `Property RaiseExceptionOnError : Boolean`

Visibility: published

Access: Read,Write

Description: `RaiseExceptionOnError` determines whether an error during a logging operation will be signaled with an exception or not. If set to `False`, errors will be silently ignored, thus not disturbing normal operation of the program.

12.5.16 TEventLog.DefaultEventType

Synopsis: Default event type for the Log (204) call.

Declaration: `Property DefaultEventType : TEventType`

Visibility: published

Access: Read,Write

Description: `DefaultEventType` is the event type used by the Log (204) call if it's `EventType` parameter is omitted.

See also: TEventLog.Log (204)

12.5.17 TEventLog.FileName

Synopsis: File name for log file

Declaration: `Property FileName : String`

Visibility: published

Access: Read,Write

Description: `FileName` is the name of the log file used to log messages if no system logger is available or the LogType (201) is `ltFile`. If none is specified, then the name of the application binary is used, with the extension replaced by .log. The file is then located in the /tmp directory on unix-like systems, or in the application directory for Dos/Windows like systems.

See also: TEventType.LogType (201)

12.5.18 TEventLog.TimeStampFormat

Synopsis: Format for the timestamp string

Declaration: `Property TimeStampFormat : String`

Visibility: published

Access: Read,Write

Description: `TimeStampFormat` is the formatting string used to create a timestamp string when writing log messages to file. It should have a format suitable for the FormatDateTime (**??**) call. If it is left empty, then `yyyy-mm-dd hh:nn:ss.zzz` is used.

See also: TEventLog.Identification (205)

12.5.19 TEventLog.CustomLogType

Synopsis: Custom log type ID

Declaration: `Property CustomLogType : Word`

Visibility: published

Access: Read,Write

Description: `CustomLogType` is used in the EventTypeToString (203) to format the custom log event type string.

See also: TEventLog.EventTypeToString (203)

12.5.20 TEventLog.EventIDOffset

Synopsis: Offset for event ID messages identifiers

Declaration: `Property EventIDOffset : DWord`

Visibility: published

Access: Read,Write

Description: `EventIDOffset` is the offset for the message formatting strings in the windows resource file. This property is ignored on other platforms.

The message strings in the file registered with the RegisterMessageFile (203) call are windows resource strings. They each have a unique ID, which must be communicated to windows. In the resource file distributed by Free Pascal, the resource strings are numbered from 1000 to 1004. The actual number communicated to windows is formed by adding the ordinal value of the message's eventtype to `EventIDOffset` (which is by default 1000), which means that by default, the string numbers are:

1000 Custom event types

1001 Information event type

1002 Warning event type

1003 Error event type

1004 Debug event type

See also: TEventLog.RegisterMessageFile (203)

12.5.21 TEventLog.OnGetCustomCategory

Synopsis: Event to retrieve custom message category

Declaration: `Property OnGetCustomCategory : TLogCategoryEvent`

Visibility: published

Access: Read,Write

Description: `OnGetCustomCategory` is called on the windows platform to determine the category of a custom event type. It should return an ID which will be used by windows to look up the string which describes the message category in the file containing the resource strings.

See also: TEventLog.OnGetCustomEventID (208), TEventLog.OnGetCustomEvent (209)

12.5.22 TEventLog.OnGetCustomEventID

Synopsis: Event to retrieve custom event ID

Declaration: `Property OnGetCustomEventID : TLogCodeEvent`

Visibility: published

Access: Read,Write

Description: `OnGetCustomEventID` is called on the windows platform to determine the category of a custom event type. It should return an ID which will be used by windows to look up the string which formats the message, in the file containing the resource strings.

See also: TEventLog.OnGetCustomCategory (208), TEventLog.OnGetCustomEvent (209)

12.5.23 TEventLog.OnGetCustomEvent

Synopsis: Event to retrieve custom event Code

Declaration: `Property OnGetCustomEvent : TLogCodeEvent`

Visibility: published

Access: Read,Write

Description: `OnGetCustomEvent` is called on the windows platform to determine the event code of a custom event type. It should return an ID.

See also: TEventLog.OnGetCustomCategory (208), TEventLog.OnGetCustomEventID (208)

Chapter 13

Reference for unit 'ezcgi'

13.1 Used units

Table 13.1: Used units by unit 'ezcgi'

Name	Page
Classes	??
strings	210
sysutils	??

13.2 Overview

ezcgi, written by Michael Hess, provides a single class which offers simple access to the CGI environment which a CGI program operates under. It supports both GET and POST methods. It's intended for simple CGI programs which do not need full-blown CGI support. File uploads are not supported by this component.

To use the unit, a descendent of the `TEZCGI` class should be created and the DoPost (213) or DoGet (213) methods should be overidden.

13.3 Constants, types and variables

13.3.1 Constants

```
hexTable = '0123456789ABCDEF'
```

String constant used to convert a number to a hexadecimal code or back.

13.4 ECGIException

13.4.1 Description

Exception raised by TEZcgi (211)

13.5 TEZcgi

13.5.1 Description

`TEZcgi` implements all functionality to analyze the CGI environment and query the variables present in it. It's main use is the exposed variables.

Programs wishing to use this class should make a descendent class of this class and override the DoPost (213) or DoGet (213) methods. To run the program, an instance of this class must be created, and it's Run (212) method should be invoked. This will analyze the environment and call the `DoPost` or `DoGet` method, depending on what HTTP method was used to invoke the program.

13.5.2 Method overview

Page	Property	Description
211	Create	Creates a new instance of the `TEZCGI` component
211	Destroy	Removes the `TEZCGI` component from memory
213	DoGet	Method to handle `GET` requests
213	DoPost	Method to handle `POST` requests
213	GetValue	Return the value of a request variable.
212	PutLine	Send a line of output to the web-client
212	Run	Run the CGI application.
212	WriteContent	Writes the content type to standard output

13.5.3 Property overview

Page	Property	Access	Description
215	Email	rw	Email of the server administrator
215	Name	rw	Name of the server administrator
214	Names	r	Indexed array with available variable names.
213	Values	r	Variables passed to the CGI script
215	VariableCount	r	Number of available variables.
214	Variables	r	Indexed array with variables as name=value pairs.

13.5.4 TEZcgi.Create

Synopsis: Creates a new instance of the `TEZCGI` component

Declaration: `constructor Create`

Visibility: public

Description: `Create` initializes the CGI program's environment: it reads the environment variables passed to the CGI program and stores them in the Variable (210) property.

See also: TZECGI.Variables (210), TZECGI.Names (210), TZECGI.Values (210)

13.5.5 TEZcgi.Destroy

Synopsis: Removes the `TEZCGI` component from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` removes all variables from memory and then calls the inherited destroy, removing the `TEZCGI` instance from memory.

`Destroy` should never be called directly. Instead `Free` should be used, or `FreeAndNil`

See also: TEZcgi.Create (211)

13.5.6 TEZcgi.Run

Synopsis: Run the CGI application.

Declaration: `procedure Run`

Visibility: public

Description: `Run` analyses the variables passed to the application, processes the request variables (it stores them in the Variables (210) property) and calls the DoPost (213) or DoGet (213) methods, depending on the method passed to the web server.

After creating the instance of `TEZCGI`, the `Run` method is the only method that should be called when using this component.

See also: TZECGI.Variables (210), TEZCGI.DoPost (213), TEZCGI.DoGet (213)

13.5.7 TEZcgi.WriteContent

Synopsis: Writes the content type to standard output

Declaration: `procedure WriteContent(ctype: String)`

Visibility: public

Description: `WriteContent` writes the content type `cType` to standard output, followed by an empty line. After this method was called, no more HTTP headers may be written to standard output. Any HTTP headers should be written before `WriteContent` is called. It should be called from the DoPost (213) or DoGet (213) methods.

See also: TEZCGI.DoPost (213), TEZCGI.DoGet (213), TEZcgi.PutLine (212)

13.5.8 TEZcgi.PutLine

Synopsis: Send a line of output to the web-client

Declaration: `procedure PutLine(sOut: String)`

Visibility: public

Description: `PutLine` writes a line of text (`sOut`) to the web client (currently, to standard output). It should be called only after WriteContent (212) was called with a content type of `text`. The sent text is not processed in any way, i.e. no HTML entities or so are inserted instead of special HTML characters. This should be done by the user.

Errors: No check is performed whether the content type is right.

See also: TEZcgi.WriteContent (212)

13.5.9 TEZcgi.GetValue

Synopsis: Return the value of a request variable.

Declaration: `function GetValue(Index: String;defaultValue: String) : String`

Visibility: public

Description: `GetValue` returns the value of the variable named `Index`, and returns `DefaultValue` if it is empty or does not exist.

See also: TEZCGI.Values (213)

13.5.10 TEZcgi.DoPost

Synopsis: Method to handle `POST` requests

Declaration: `procedure DoPost;  Virtual`

Visibility: public

Description: `DoPost` is called by the Run (212) method the `POST` method was used to invoke the CGI application. It should be overridden in descendents of `TEZcgi` to actually handle the request.

See also: TEZcgi.Run (212), TEZcgi.DoGet (213)

13.5.11 TEZcgi.DoGet

Synopsis: Method to handle `GET` requests

Declaration: `procedure DoGet;  Virtual`

Visibility: public

Description: `DoGet` is called by the Run (212) method the `GET` method was used to invoke the CGI application. It should be overridden in descendents of `TEZcgi` to actually handle the request.

See also: TEZcgi.Run (212), TEZcgi.DoPost (213)

13.5.12 TEZcgi.Values

Synopsis: Variables passed to the CGI script

Declaration: `Property Values[Index: String]: String`

Visibility: public

Access: Read

Description: `Values` is a name-based array of variables that were passed to the script by the web server or the HTTP request. The `Index` variable is the name of the variable whose value should be retrieved. The following standard values are available:

AUTH_TYPE Authorization type

CONTENT_LENGTH Content length

CONTENT_TYPE Content type

GATEWAY_INTERFACE Used gateway interface

PATH_INFO Requested URL

PATH_TRANSLATED Transformed URL

QUERY_STRING Client query string

REMOTE_ADDR Address of remote client

REMOTE_HOSTDNS name of remote client

REMOTE_IDENT Remote identity.

REMOTE_USER Remote user

REQUEST_METHOD Request methods (POST or GET)

SCRIPT_NAME Script name

SERVER_NAME Server host name

SERVER_PORT Server port

SERVER_PROTOCOL Server protocol

SERVER_SOFTWARE Web server software

HTTP_ACCEPT Accepted responses

HTTP_ACCEPT_CHARSET Accepted character sets

HTTP_ACCEPT_ENCODING Accepted encodings

HTTP_IF_MODIFIED_SINCE Proxy information

HTTP_REFERER Referring page

HTTP_USER_AGENT Client software name

Other than the standard list, any variables that were passed by the web-client request, are also available. Note that the variables are case insensitive.

See also: TEZCGI.Variables (214), TEZCGI.Names (214), TEZCGI.GetValue (213), TEZcgi.VariableCount (215)

13.5.13 TEZcgi.Names

Synopsis: Indexed array with available variable names.

Declaration: `Property Names[Index: Integer]: String`

Visibility: public

Access: Read

Description: `Names` provides indexed access to the available variable names. The `Index` may run from 0 to VariableCount (215). Any other value will result in an exception being raised.

See also: TEZcgi.Variables (214), TEZcgi.Values (213), TEZcgi.GetValue (213), TEZcgi.VariableCount (215)

13.5.14 TEZcgi.Variables

Synopsis: Indexed array with variables as name=value pairs.

Declaration: `Property Variables[Index: Integer]: String`

Visibility: public

Access: Read

Description: `Variables` provides indexed access to the available variable names and values. The variables are returned as `Name=Value` pairs. The `Index` may run from 0 to VariableCount (215). Any other value will result in an exception being raised.

See also: TEZcgi.Names (214), TEZcgi.Values (213), TEZcgi.GetValue (213), TEZcgi.VariableCount (215)

13.5.15 TEZcgi.VariableCount

Synopsis: Number of available variables.

Declaration: `Property VariableCount : Integer`

Visibility: public

Access: Read

Description: `TEZcgi.VariableCount` returns the number of available CGI variables. This includes both the standard CGI environment variables and the request variables. The actual names and values can be retrieved with the Names (214) and Variables (214) properties.

See also: TEZcgi.Names (214), TEZcgi.Variables (214), TEZcgi.Values (213), TEZcgi.GetValue (213)

13.5.16 TEZcgi.Name

Synopsis: Name of the server administrator

Declaration: `Property Name : String`

Visibility: public

Access: Read,Write

Description: `Name` is used when displaying an error message to the user. This should set prior to calling the TEZcgi.Run (212) method.

See also: TEZcgi.Run (212), TEZcgi.Email (215)

13.5.17 TEZcgi.Email

Synopsis: Email of the server administrator

Declaration: `Property Email : String`

Visibility: public

Access: Read,Write

Description: `Email` is used when displaying an error message to the user. This should set prior to calling the TEZcgi.Run (212) method.

See also: TEZcgi.Run (212), TEZcgi.Name (215)

Chapter 14

Reference for unit 'fpTimer'

14.1 Used units

Table 14.1: Used units by unit 'fpTimer'

Name	Page
Classes	??

14.2 Overview

The fpTimer unit implements a timer class TFPTimer (218) which can be used on all supported platforms. The timer class uses a driver class TFPTimerDriver (219) which does the actual work.

A default timer driver class is implemented on all platforms. It will work in GUI and non-gui applications, but only in the application's main thread.

An alternative driver class can be used by setting the DefaultTimerDriverClass (216) variable to the class pointer of the driver class. The driver class should descend from TFPTimerDriver (219).

14.3 Constants, types and variables

14.3.1 Types

```
TFPTimerDriverClass = Class of TFPTimerDriver
```

`TFPTimerDriverClass` is the class pointer of TFPTimerDriver (219) it exists mainly for the purpose of being able to set DefaultTimerDriverClass (216), so a custom timer driver can be used for the timer instances.

14.3.2 Variables

```
DefaultTimerDriverClass : TFPTimerDriverClass = nil
```

DefaultTimerDriverClass contains the TFPTimerDriver (219) class pointer that should be used when a new instance of TFPCustomTimer (217) is created. It is by default set to the system timer class.

Setting this class pointer to another descendent of TFPTimerDriver allows to customize the default timer implementation used in the entire application.

14.4 TFPCustomTimer

14.4.1 Description

TFPCustomTimer is the timer class containing the timer's implementation. It relies on an extra driver instance (of type TFPTimerDriver (219)) to do the actual work.

TFPCustomTimer publishes no events or properties, so it is unsuitable for handling in an IDE. The TFPTimer (218) descendent class publishes all needed events of TFPCustomTimer.

14.4.2 Method overview

Page	Property	Description
217	Create	Create a new timer
217	Destroy	Release a timer instance from memory
218	StartTimer	Start the timer
218	StopTimer	Stop the timer

14.4.3 TFPCustomTimer.Create

Synopsis: Create a new timer

Declaration: `constructor Create(AOwner: TComponent);  Override`

Visibility: public

Description: Create instantiates a new TFPCustomTimer instance. It creates the timer driver instance from the DefaultTimerDriverClass class pointer.

See also: TFPCustomTimer.Destroy (217)

14.4.4 TFPCustomTimer.Destroy

Synopsis: Release a timer instance from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: Destroy releases the timer driver component from memory, and then calls Inherited to clean the TFPCustomTimer instance from memory.

See also: TFPCustomTimer.Create (217)

14.4.5 TFPCustomTimer.StartTimer

Synopsis: Start the timer

Declaration: `procedure StartTimer;  Virtual`

Visibility: public

Description: `StartTimer` starts the timer. After a call to `StartTimer`, the timer will start producing timer ticks.

The timer stops producing ticks only when the StopTimer (218) event is called.

See also: TFPCustomTimer.StopTimer (218), TFPTimer.Enabled (218), TFPTimer.OnTimer (219)

14.4.6 TFPCustomTimer.StopTimer

Synopsis: Stop the timer

Declaration: `procedure StopTimer;  Virtual`

Visibility: public

Description: `Stoptimer` stops a started timer. After a call to `StopTimer`, the timer no longer produces timer ticks.

See also: TFPCustomTimer.StartTimer (218), TFPTimer.Enabled (218), TFPTimer.OnTimer (219)

14.5 TFPTimer

14.5.1 Description

`TFPTimer` implements no new events or properties, but merely publishes events and properties already implemented in TFPCustomTimer (217): Enabled (218), OnTimer (219) and Interval (219).

The `TFPTimer` class is suitable for use in an IDE.

14.5.2 Property overview

Page	Property	Access	Description
218	Enabled		Start or stop the timer
219	Interval		Timer tick interval in milliseconds.
219	OnTimer		Event called on each timer tick.

14.5.3 TFPTimer.Enabled

Synopsis: Start or stop the timer

Declaration: `Property Enabled :`

Visibility: published

Access:

Description: `Enabled` controls whether the timer is active. Setting `Enabled` to `True` will start the timer (calling StartTimer (218)), setting it to `False` will stop the timer (calling StopTimer (218)).

See also: TFPCustomTimer.StartTimer (218), TFPCustomTimer.StopTimer (218), TFPTimer.OnTimer (219), TFPTimer.Interval (219)

14.5.4 TFPTimer.Interval

Synopsis: Timer tick interval in milliseconds.

Declaration: `Property Interval :`

Visibility: published

Access:

Description: `Interval` specifies the timer interval in milliseconds. Every `Interval` milliseconds, the OnTimer (219) event handler will be called.

Note that the milliseconds interval is a minimum interval. Under high system load, the timer tick may arrive later.

See also: TFPTimer.OnTimer (219), TFPTimer.Enabled (218)

14.5.5 TFPTimer.OnTimer

Synopsis: Event called on each timer tick.

Declaration: `Property OnTimer :`

Visibility: published

Access:

Description: `OnTimer` is called on each timer tick. The event handler must be assigned to a method that will do the actual work that should occur when the timer fires.

See also: TFPTimer.Interval (219), TFPTimer.Enabled (218)

14.6 TFPTimerDriver

14.6.1 Description

`TFPTimerDriver` is the abstract timer driver class: it simply provides an interface for the TFPCustomTimer (217) class to use.

The `fpTimer` unit implements a descendent of this class which implements the default timer mechanism.

14.6.2 Method overview

Page	Property	Description
220	Create	Create new driver instance
220	StartTimer	Start the timer
220	StopTimer	Stop the timer

14.6.3 Property overview

Page	Property	Access	Description
220	Timer	r	Timer tick

14.6.4 TFPTimerDriver.Create

Synopsis: Create new driver instance

Declaration: `constructor Create(ATimer: TFPCustomTimer); Virtual`

Visibility: public

Description: `Create` should be overridden by descendents of `TFPTimerDriver` to do additional initialization of the timer driver. `Create` just stores (in Timer (220)) a reference to the `ATimer` instance which created the driver instance.

See also: TFPTimerDriver.Timer (220), TFPTimer (218)

14.6.5 TFPTimerDriver.StartTimer

Synopsis: Start the timer

Declaration: `procedure StartTimer; Virtual; Abstract`

Visibility: public

Description: `StartTimer` is called by TFPCustomTimer.StartTimer (218). It should be overridden by descendents of `TFPTimerDriver` to actually start the timer.

See also: TFPCustomTimer.StartTimer (218), TFPTimerDriver.StopTimer (220)

14.6.6 TFPTimerDriver.StopTimer

Synopsis: Stop the timer

Declaration: `procedure StopTimer; Virtual; Abstract`

Visibility: public

Description: `StopTimer` is called by TFPCustomTimer.StopTimer (218). It should be overridden by descendents of `TFPTimerDriver` to actually stop the timer.

See also: TFPCustomTimer.StopTimer (218), TFPTimerDriver.StartTimer (220)

14.6.7 TFPTimerDriver.Timer

Synopsis: Timer tick

Declaration: `Property Timer : TFPCustomTimer`

Visibility: public

Access: Read

Description: `Timer` calls the TFPCustomTimer (217) timer event. Descendents of `TFPTimerDriver` should call `Timer` whenever a timer tick occurs.

See also: TFPTimer.OnTimer (219), TFPTimerDriver.StartTimer (220), TFPTimerDriver.StopTimer (220)

Chapter 15

Reference for unit 'gettext'

15.1 Used units

Table 15.1: Used units by unit 'gettext'

Name	Page
Classes	??
sysutils	??

15.2 Overview

The gettext unit can be used to hook into the resource string mechanism of Free Pascal to provide translations of the resource strings, based on the GNU gettext mechanism. The unit provides a class (TMOFile (223)) to read the .mo files with localizations for various languages. It also provides a couple of calls to translate all resource strings in an application based on the translations in a .mo file.

15.3 Constants, types and variables

15.3.1 Constants

```
MOFileHeaderMagic = $950412de
```

This constant is found as the first integer in a .mo

15.3.2 Types

```
PLongWordArray = ^TLongWordArray
```

Pointer to a TLongWordArray (222) array.

```
PMOStringTable = ^TMOStringTable
```

Pointer to a TMOStringTable (222) array.

```
PPCharArray = ^TPCharArray
```

Pointer to a TPCharArray (222) array.

```
TLongWordArray = Array[0..(1shl30)divSizeOf(LongWord)] of LongWord
```

TLongWordArray is an array used to define the PLongWordArray (221) pointer. A variable of type TLongWordArray should never be directly declared, as it would occupy too much memory. The PLongWordArray type can be used to allocate a dynamic number of elements.

```
TMOFileHeader = packed record
  magic : LongWord;
  revision : LongWord;
  nstrings : LongWord;
  OrigTabOffset : LongWord;
  TransTabOffset : LongWord;
  HashTabSize : LongWord;
  HashTabOffset : LongWord;
end
```

This structure describes the structure of a .mo file with string localizations.

```
TMOStringInfo = packed record
  length : LongWord;
  offset : LongWord;
end
```

This record is one element in the string tables describing the original and translated strings. It describes the position and length of the string. The location of these tables is stored in the TMOFileHeader (222) record at the start of the file.

```
TMOStringTable = Array[0..(1shl30)divSizeOf(TMOStringInfo)] of TMOStringInfo
```

TMOStringTable is an array type containing TMOStringInfo (222) records. It should never be used directly, as it would occupy too much memory.

```
TPCharArray = Array[0..(1shl30)divSizeOf(PChar)] of PChar
```

TLongWordArray is an array used to define the PPCharArray (222) pointer. A variable of type TPCharArray should never be directly declared, as it would occupy too much memory. The PPCharArray type can be used to allocate a dynamic number of elements.

15.4 Procedures and functions

15.4.1 GetLanguageIDs

Synopsis: Return the current language IDs

Declaration: `procedure GetLanguageIDs(var Lang: String;var FallbackLang: String)`

Visibility: default

Description: `GetLanguageIDs` returns the current language IDs (an ISO string) as returned by the operating system. On windows, the `GetUserDefaultLCID` and `GetLocaleInfo` calls are used. On other operating systems, the `LC_ALL`, `LC_MESSAGES` or `LANG` environment variables are examined.

15.4.2 TranslateResourceStrings

Synopsis: Translate the resource strings of the application.

Declaration:
```
procedure TranslateResourceStrings(AFile: TMOFile)
procedure TranslateResourceStrings(const AFilename: String)
```

Visibility: default

Description: `TranslateResourceStrings` translates all the resource strings in the application based on the values in the .mo file `AFileName` or `AFile`. The procedure creates an TMOFile (223) instance to read the .mo file if a filename is given.

Errors: If the file does not exist or is an invalid .mo file.

See also: TranslateUnitResourceStrings (223), TMOFile (223)

15.4.3 TranslateUnitResourceStrings

Synopsis: Translate the resource strings of a unit.

Declaration:
```
procedure TranslateUnitResourceStrings(const AUnitName: String;
                                       AFile: TMOFile)
procedure TranslateUnitResourceStrings(const AUnitName: String;
                                       const AFilename: String)
```

Visibility: default

Description: `TranslateUnitResourceStrings` is identical in function to TranslateResourceStrings (223), but translates the strings of a single unit (`AUnitName`) which was used to compile the application. This can be more convenient, since the resource string files are created on a unit basis.

See also: TranslateResourceStrings (223), TMOFile (223)

15.5 EMOFileError

15.5.1 Description

`EMOFileError` is raised in case an TMOFile (223) instance is created with an invalid .mo.

15.6 TMOFile

15.6.1 Description

`TMOFile` is a class providing easy access to a .mo file. It can be used to translate any of the strings that reside in the .mo file. The internal structure of the .mo is completely hidden.

15.6.2 Method overview

Page	Property	Description
224	Create	Create a new instance of the `TMOFile` class.
224	Destroy	Removes the `TMOFile` instance from memory
224	Translate	Translate a string

15.6.3 TMOFile.Create

Synopsis: Create a new instance of the `TMOFile` class.

Declaration:
```
constructor Create(const AFilename: String)
constructor Create(AStream: TStream)
```

Visibility: public

Description: `Create` creates a new instance of the `MOFile` class. It opens the file `AFileName` or the stream `AStream`. If a stream is provided, it should be seekable.

The whole contents of the file is read into memory during the `Create` call. This means that the stream is no longer needed after the `Create` call.

Errors: If the named file does not exist, then an exception may be raised. If the file does not contain a valid TMOFileHeader (222) structure, then an EMOFileError (223) exception is raised.

See also: TMOFile.Destroy (224)

15.6.4 TMOFile.Destroy

Synopsis: Removes the `TMOFile` instance from memory

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` cleans the internal structures with the contents of the .mo. After this the `TMOFile` instance is removed from memory.

See also: TMOFile.Create (224)

15.6.5 TMOFile.Translate

Synopsis: Translate a string

Declaration:
```
function Translate(AOrig: PChar;ALen: Integer;AHash: LongWord) : String
function Translate(AOrig: String;AHash: LongWord) : String
function Translate(AOrig: String) : String
```

Visibility: public

Description: `Translate` translates the string `AOrig`. The string should be in the .mo file as-is. The string can be given as a plain string, as a `PChar` (with length `ALen`). If the hash value (`AHash`) of the string is not given, it is calculated.

If the string is in the `.mo` file, the translated string is returned. If the string is not in the file, an empty string is returned.

Errors: None.

Chapter 16

Reference for unit 'idea'

16.1 Used units

Table 16.1: Used units by unit 'idea'

Name	Page
Classes	**??**
sysutils	**??**

16.2 Overview

Besides some low level IDEA encryption routines, the IDEA unit also offers 2 streams which offer on-the-fly encryption or decryption: there are 2 stream objects: A write-only encryption stream which encrypts anything that is written to it, and a decription stream which decrypts anything that is read from it.

16.3 Constants, types and variables

16.3.1 Constants

```
IDEABLOCKSIZE = 8
```

IDEA block size

```
IDEAKEYSIZE = 16
```

IDEA Key size constant.

```
KEYLEN = (6 * ROUNDS + 4 )
```

Key length

```
ROUNDS = 8
```

Number of rounds to encrypt

16.3.2 Types

```
IdeaCryptData = TIdeaCryptData
```

Provided for backward functionality.

```
IdeaCryptKey = TIdeaCryptKey
```

Provided for backward functionality.

```
IDEAkey = TIDEAKey
```

Provided for backward functionality.

```
TIdeaCryptData = Array[0..3] of Word
```

`TIdeaCryptData` is an internal type, defined to hold data for encryption/decryption.

```
TIdeaCryptKey = Array[0..7] of Word
```

The actual encryption or decryption key for IDEA is 64-bit long. This type is used to hold such a key. It can be generated with the EnKeyIDEA (227) or DeKeyIDEA (226) algorithms depending on whether an encryption or decryption key is needed.

```
TIDEAKey = Array[0..keylen-1] of Word
```

The IDEA key should be filled by the user with some random data (say, a passphrase). This key is used to generate the actual encryption/decryption keys.

16.4 Procedures and functions

16.4.1 CipherIdea

Synopsis: Encrypt or decrypt a buffer.

Declaration:
```
procedure CipherIdea(Input: TIdeaCryptData;var outdata: TIdeaCryptData;
                     z: TIDEAKey)
```

Visibility: default

Description: `CipherIdea` encrypts or decrypts a buffer with data (`Input`) using key `z`. The resulting encrypted or decrypted data is returned in `Output`.

Errors: None.

See also: EnKeyIdea (227), DeKeyIdea (226), TIDEAEncryptStream (228), TIDEADecryptStream (227)

16.4.2 DeKeyIdea

Synopsis: Create a decryption key from an encryption key.

Declaration:
```
procedure DeKeyIdea(z: TIDEAKey;var dk: TIDEAKey)
```

Visibility: default

Description: `DeKeyIdea` creates a decryption key based on the encryption key `z`. The decryption key is returned in `dk`. Note that only a decryption key generated from the encryption key that was used to encrypt the data can be used to decrypt the data.

Errors: None.

See also: EnKeyIdea (227), CipherIdea (226)

16.4.3 EnKeyIdea

Synopsis: Create an IDEA encryption key from a user key.

Declaration: `procedure EnKeyIdea(UserKey: TIdeaCryptKey;var z: TIDEAKey)`

Visibility: default

Description: `EnKeyIdea` creates an IDEA encryption key from user-supplied data in `UserKey`. The Encryption key is stored in `z`.

Errors: None.

See also: DeKeyIdea (226), CipherIdea (226)

16.5 EIDEAError

16.5.1 Description

`EIDEAError` is used to signal errors in the IDEA encryption decryption streams.

16.6 TIDEADeCryptStream

16.6.1 Description

`TIDEADecryptStream` is a stream which decrypts anything that is read from it using the IDEA mechanism. It reads the encrypted data from a source stream and decrypts it using the CipherIDEA (226) algorithm. It is a read-only stream: it is not possible to write data to this stream.

When creating a `TIDEADecryptStream` instance, an IDEA decryption key should be passed to the constructor, as well as the stream from which encrypted data should be read written.

The encrypted data can be created with a TIDEAEncryptStream (228) encryption stream.

16.6.2 Method overview

Page	Property	Description
227	Create	Constructor to create a new `TIDEADecryptStream` instance
228	Read	Reads data from the stream, decrypting it as needed
228	Seek	Set position on the stream

16.6.3 TIDEADeCryptStream.Create

Synopsis: Constructor to create a new `TIDEADecryptStream` instance

Declaration: `constructor Create(const AKey: String;Dest: TStream);  Overload`

Visibility: public

Description: `Create` creates a new `TIDEADecryptStream` instance using the the string `AKey` to compute the encryption key (226), which is then passed on to the inherited constructor TIDEAStream.Create (231). It is an easy-access function which introduces no new functionality.

The string is truncated at the maximum length of the TIdeaCryptKey (226) structure, so it makes no sense to provide a string with length longer than this structure.

See also: TIdeaCryptKey (226), TIDEAStream.Create (231), TIDEAEnCryptStream.Create (229)

16.6.4 TIDEADeCryptStream.Read

Synopsis: Reads data from the stream, decrypting it as needed

Declaration: `function Read(var Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` attempts to read `Count` bytes from the stream, placing them in `Buffer` the bytes are read from the source stream and decrypted as they are read. (bytes are read from the source stream in blocks of 8 bytes. The function returns the number of bytes actually read.

Errors: If an error occurs when reading data from the source stream, an exception may be raised.

See also: TIDEADecryptStream.Write (227), TIDEADecryptStream.Seek (228), TIDEAEncryptStream (228)

16.6.5 TIDEADeCryptStream.Seek

Synopsis: Set position on the stream

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` will only work on a forward seek. It emulates a forward seek by reading and discarding bytes from the input stream. The `TIDEADecryptStream` stream tries to provide seek capabilities for the following limited number of cases:

Origin=soFromBeginning If `Offset` is larger than the current position, then the remaining bytes are skipped by reading them from the stream and discarding them.

Origin=soFromCurrent If `Offset` is zero, the current position is returned. If it is positive, then `Offset` bytes are skipped by reading them from the stream and discarding them.

Errors: An EIDEAError (227) exception is raised if the stream does not allow the requested seek operation.

See also: TIDEADeCryptStream.Read (228)

16.7 TIDEAEncryptStream

16.7.1 Description

`TIDEAEncryptStream` is a stream which encrypts anything that is written to it using the IDEA mechanism, and then writes the encrypted data to the destination stream using the CipherIDEA (226) algorithm. It is a write-only stream: it is not possible to read data from this stream.

When creating a `TIDEAEncryptStream` instance, an IDEA encryption key should be passed to the constructor, as well as the stream to which encrypted data should be written.

The resulting encrypted data can be read again with a TIDEADecryptStream (227) decryption stream.

16.7.2 Method overview

Page	Property	Description
229	Create	Constructor to create a new `TIDEAEncryptStream` instance
229	Destroy	Flush data buffers and free the stream instance.
230	Flush	Write remaining bytes from the stream
230	Seek	Set stream position
229	Write	Write bytes to the stream to be encrypted

16.7.3 TIDEAEncryptStream.Create

Synopsis: Constructor to create a new `TIDEAEncryptStream` instance

Declaration:
```
constructor Create(const AKey: String;Dest: TStream);  Overload
```

Visibility: public

Description: `Create` creates a new `TIDEAEncryptStream` instance using the the string `AKey` to compute the encryption key (226), which is then passed on to the inherited constructor TIDEAStream.Create (231). It is an easy-access function which introduces no new functionality.

The string is truncated at the maximum length of the TIdeaCryptKey (226) structure, so it makes no sense to provide a string with length longer than this structure.

See also: TIdeaCryptKey (226), TIDEAStream.Create (231), TIDEADeCryptStream.Create (227)

16.7.4 TIDEAEncryptStream.Destroy

Synopsis: Flush data buffers and free the stream instance.

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` flushes any data still remaining in the internal encryption buffer, and then calls the inherited `Destroy`

By default, the destination stream is not freed when the encryption stream is freed.

Errors: None.

See also: TIDEAStream.Create (231)

16.7.5 TIDEAEncryptStream.Write

Synopsis: Write bytes to the stream to be encrypted

Declaration:
```
function Write(const Buffer;Count: LongInt) : LongInt;  Override
```

Visibility: public

Description: `Write` writes `Count` bytes from `Buffer` to the stream, encrypting the bytes as they are written (encryption in blocks of 8 bytes).

Errors: If an error occurs writing to the destination stream, an error may occur.

See also: TIDEADecryptStream.Read (228)

16.7.6 TIDEAEncryptStream.Seek

Synopsis: Set stream position

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` return the current position if called with `0` and `soFromCurrent` as arguments. With all other values, it will always raise an exception, since it is impossible to set the position on an encryption stream.

Errors: An EIDEAError (227) will be raised unless called with `0` and `soFromCurrent` as arguments.

See also: TIDEAEncryptStream.Write (229), EIDEAError (227)

16.7.7 TIDEAEncryptStream.Flush

Synopsis: Write remaining bytes from the stream

Declaration: `procedure Flush`

Visibility: public

Description: `Flush` writes the current encryption buffer to the stream. Encryption always happens in blocks of 8 bytes, so if the buffer is not completely filled at the end of the writing operations, it must be flushed. It should never be called directly, unless at the end of all writing operations. It is called automatically when the stream is destroyed.

Errors: None.

See also: TIDEAEncryptStream.Write (229)

16.8 TIDEAStream

16.8.1 Description

Do not create instances of `TIDEAStream` directly. It implements no useful functionality: it serves as a common ancestor of the TIDEAEncryptStream (228) and TIDEADeCryptStream (227), and simply provides some fields that these descendent classes use when encrypting/decrypting. One of these classes should be created, depending on whether one wishes to encrypt or to decrypt.

16.8.2 Method overview

Page	Property	Description
231	Create	Creates a new instance of the `TIDEAStream` class

16.8.3 Property overview

Page	Property	Access	Description
231	Key	r	Key used when encrypting/decrypting

16.8.4 TIDEAStream.Create

Synopsis: Creates a new instance of the `TIDEAStream` class

Declaration: `constructor Create(AKey: TIDEAKey;Dest: TStream); Overload`

Visibility: public

Description: `Create` stores the encryption/decryption key and then calls the inherited `Create` to store the `Dest` stream.

Errors: None.

See also: TIDEAEncryptStream (228), TIDEADeCryptStream (227)

16.8.5 TIDEAStream.Key

Synopsis: Key used when encrypting/decrypting

Declaration: `Property Key : TIDEAKey`

Visibility: public

Access: Read

Description: `Key` is the key as it was passed to the constructor of the stream. It cannot be changed while data is read or written. It is the key as it is used when encrypting/decrypting.

See also: CipherIdea (226)

Chapter 17

Reference for unit 'inicol'

17.1 Used units

Table 17.1: Used units by unit 'inicol'

Name	Page
Classes	??
Inifiles	232
sysutils	??

17.2 Overview

inicol contains an implementation of `TCollection` and `TCollectionItem` descendents which cooperate to read and write the collection from and to a .ini file. It uses the TCustomIniFile (242) class for this.

17.3 Constants, types and variables

17.3.1 Constants

```
KeyCount = 'Count'
```

`KeyCount` is used as a key name when reading or writing the number of items in the collection from the global section.

```
SGlobal = 'Global'
```

`SGlobal` is used as the default name of the global section when reading or writing the collection.

17.4 EIniCol

17.4.1 Description

`EIniCol` is used to report error conditions in the load and save methods of TIniCollection (233).

17.5 TIniCollection

17.5.1 Description

`TIniCollection` is a collection (**??**) descendent which has the capability to write itself to an .ini file. It introduces some load and save mechanisms, which can be used to write all items in the collection to disk. The items should be descendents of the type TIniCollectionItem (236).

All methods work using a `TCustomInifile` class, making it possible to save to alternate file formats, or even databases.

An instance of `TIniCollection` should never be used directly. Instead, a descendent should be used, which sets the `FPrefix` and `FSectionPrefix` protected variables.

17.5.2 Method overview

Page	Property	Description
233	Load	Loads the collection from the default filename.
235	LoadFromFile	Load collection from file.
235	LoadFromIni	Load collection from a file in .ini file format.
234	Save	Save the collection to the default filename.
234	SaveToFile	Save collection to a file in .ini file format
234	SaveToIni	Save the collection to a `TCustomIniFile` descendent

17.5.3 Property overview

Page	Property	Access	Description
236	FileName	rw	Filename of the collection
236	GlobalSection	rw	Name of the global section
235	Prefix	r	Prefix used in global section
236	SectionPrefix	r	Prefix string for section names

17.5.4 TIniCollection.Load

Synopsis: Loads the collection from the default filename.

Declaration: `procedure Load`

Visibility: public

Description: `Load` loads the collection from the file as specified in the FileName (236) property. It calls the LoadFromFile (235) method to do this.

Errors: If the collection was not loaded or saved to file before this call, an `EIniCol` exception will be raised.

See also: TIniCollection.LoadFromFile (235), TIniCollection.LoadFromIni (235), TIniCollection.Save (234), TIniCollection.FileName (236)

17.5.5 TIniCollection.Save

Synopsis: Save the collection to the default filename.

Declaration: `procedure Save`

Visibility: public

Description: `Save` writes the collection to the file as specified in the FileName (236) property, using GlobalSection (236) as the section. It calls the SaveToFile (234) method to do this.

Errors: If the collection was not loaded or saved to file before this call, an `EIniCol` exception will be raised.

See also: TIniCollection.SaveToFile (234), TIniCollection.SaveToIni (234), TIniCollection.Load (233), TIniCollection.FileName (236)

17.5.6 TIniCollection.SaveToIni

Synopsis: Save the collection to a `TCustomIniFile` descendent

Declaration: `procedure SaveToIni(Ini: TCustomInifile;Section: String);  Virtual`

Visibility: public

Description: `SaveToIni` does the actual writing. It writes the number of elements in the global section (as specified by the `Section` argument), as well as the section name for each item in the list. The item names are written using the Prefix (235) property for the key. After this it calls the SaveToIni (237) method of all TIniCollectionItem (236) instances.

This means that the global section of the .ini file will look something like this:

```
[globalsection]
Count=3
Prefix1=SectionPrefixFirstItemName
Prefix2=SectionPrefixSecondItemName
Prefix3=SectionPrefixThirdItemName
```

This construct allows to re-use an ini file for multiple collections.

After this method is called, the GlobalSection (236) property contains the value of `Section`, it will be used in the Save (236) method.

See also: TIniCollectionItem.SaveToIni (237)

17.5.7 TIniCollection.SaveToFile

Synopsis: Save collection to a file in .ini file format

Declaration: `procedure SaveToFile(AFileName: String;Section: String)`

Visibility: public

Description: `SaveToFile` will create a `TMemIniFile` instance with the `AFileName` argument as a filename. This instance is passed on to the SaveToIni (234) method, together with the `Section` argument, to do the actual saving.

Errors: An exception may be raised if the path in `AFileName` does not exist.

See also: TIniCollection.SaveToIni (234), TIniCollection.LoadFromFile (235)

17.5.8 TIniCollection.LoadFromIni

Synopsis: Load collection from a file in .ini file format.

Declaration: `procedure LoadFromIni(Ini: TCustomInifile;Section: String); Virtual`

Visibility: public

Description: `LoadFromIni` will load the collection from the `Ini` instance. It first clears the collection, and reads the number of items from the global section with the name as passed through the `Section` argument. After this, an item is created and added to the collection, and its data is read by calling the TIniCollectionItem.LoadFromIni (237) method, passing the appropriate section name as found in the global section.

The description of the global section can be found in the TIniCollection.SaveToIni (234) method description.

See also: TIniCollection.LoadFromFile (235), TIniCollectionItem.LoadFromIni (237), TIniCollection.SaveToIni (234)

17.5.9 TIniCollection.LoadFromFile

Synopsis: Load collection from file.

Declaration: `procedure LoadFromFile(AFileName: String;Section: String)`

Visibility: public

Description: `LoadFromFile` creates a `TMemIniFile` instance using `AFileName` as the filename. It calls LoadFromIni (235) using this instance and `Section` as the parameters.

See also: TIniCollection.LoadFromIni (235), TIniCollection.Load (233), TIniCollection.SaveToIni (234), TIniCollection.SaveToFile (234)

17.5.10 TIniCollection.Prefix

Synopsis: Prefix used in global section

Declaration: `Property Prefix : String`

Visibility: public

Access: Read

Description: `Prefix` is used when writing the section names of the items in the collection to the global section, or when reading the names from the global section. If the prefix is set to `Item` then the global section might look something like this:

```
[MyCollection]
Count=2
Item1=FirstItem
Item2=SecondItem
```

A descendent of `TIniCollection` should set the value of this property, it cannot be empty.

See also: TIniCollection.SectionPrefix (236), TIniCollection.GlobalSection (236)

17.5.11 TIniCollection.SectionPrefix

Synopsis: Prefix string for section names

Declaration: `Property SectionPrefix : String`

Visibility: public

Access: Read

Description: `SectionPrefix` is a string that is prepended to the section name as returned by the TIniCollectionItem.SectionName (238) property to return the exact section name. It can be empty.

See also: TIniCollection.Section (233), TIniCollection.GlobalSection (236)

17.5.12 TIniCollection.FileName

Synopsis: Filename of the collection

Declaration: `Property FileName : String`

Visibility: public

Access: Read,Write

Description: `FileName` is the filename as used in the last LoadFromFile (235) or SaveToFile (234) operation. It is used in the Load (233) or Save (234) calls.

See also: TIniCollection.Save (234), TIniCollection.LoadFromFile (235), TIniCollection.SaveToFile (234), TIniCollection.Load (233)

17.5.13 TIniCollection.GlobalSection

Synopsis: Name of the global section

Declaration: `Property GlobalSection : String`

Visibility: public

Access: Read,Write

Description: `GlobalSection` contains the value of the `Section` argument in the LoadFromIni (235) or SaveToIni (234) calls. It's used in the Load (233) or Save (234) calls.

See also: TIniCollection.Save (234), TIniCollection.LoadFromFile (235), TIniCollection.SaveToFile (234), TIniCollection.Load (233)

17.6 TIniCollectionItem

17.6.1 Description

`TIniCollectionItem` is a #rtl.classes.tcollectionitem (**??**) descendent which has some extra methods for saving/loading the item to or from an .ini file.

To use this class, a descendent should be made, and the SaveToIni (237) and LoadFromIni (237) methods should be overridden. They should implement the actual loading and saving. The loading and saving is always initiated by the methods in TIniCollection (233), TIniCollection.LoadFromIni (235) and TIniCollection.SaveToIni (234) respectively.

17.6.2 Method overview

Page	Property	Description
238	LoadFromFile	Load item from a file
237	LoadFromIni	Method called when the item must be loaded
237	SaveToFile	Save item to a file
237	SaveToIni	Method called when the item must be saved

17.6.3 Property overview

Page	Property	Access	Description
238	SectionName	rw	Default section name

17.6.4 TIniCollectionItem.SaveToIni

Synopsis: Method called when the item must be saved

Declaration:
```
procedure SaveToIni(Ini: TCustomInifile;Section: String);  Virtual
                        ;  Abstract
```

Visibility: public

Description: `SaveToIni` is called by TIniCollection.SaveToIni (234) when it saves this item. Descendent classes should override this method to save the data they need to save. All write methods of the `TCustomIniFile` instance passed in `Ini` can be used, as long as the writing happens in the section passed in `Section`.

Errors: No checking is done to see whether the values are actually written to the correct section.

See also: TIniCollection.SaveToIni (234), TIniCollectionItem.LoadFromIni (237), TIniCollectionItem.SaveToFile (237), TIniCollectionItem.LoadFromFile (238)

17.6.5 TIniCollectionItem.LoadFromIni

Synopsis: Method called when the item must be loaded

Declaration:
```
procedure LoadFromIni(Ini: TCustomInifile;Section: String);  Virtual
                        ;  Abstract
```

Visibility: public

Description: `LoadFromIni` is called by TIniCollection.LoadFromIni (235) when it saves this item. Descendent classes should override this method to load the data they need to load. All read methods of the `TCustomIniFile` instance passed in `Ini` can be used, as long as the reading happens in the section passed in `Section`.

Errors: No checking is done to see whether the values are actually read from the correct section.

See also: TIniCollection.LoadFromIni (235), TIniCollectionItem.SaveToIni (237), TIniCollectionItem.LoadFromFile (238), TIniCollectionItem.SaveToFile (237)

17.6.6 TIniCollectionItem.SaveToFile

Synopsis: Save item to a file

Declaration:
```
procedure SaveToFile(FileName: String;Section: String)
```

Visibility: public

Description: `SaveToFile` creates an instance of `TIniFile` with the indicated `FileName` calls SaveToIni (237) to save the item to the indicated file in .ini format under the section `Section`

Errors: An exception can occur if the file is not writeable.

See also: TIniCollectionItem.SaveToIni (237), TIniCollectionItem.LoadFromFile (238)

17.6.7 TIniCollectionItem.LoadFromFile

Synopsis: Load item from a file

Declaration: `procedure LoadFromFile(FileName: String;Section: String)`

Visibility: public

Description: `LoadFromFile` creates an instance of `TMemIniFile` and calls LoadFromIni (237) to load the item from the indicated file in .ini format from the section `Section`.

Errors: None.

See also: TIniCollectionItem.SaveToFile (237), TIniCollectionItem.LoadFromIni (237)

17.6.8 TIniCollectionItem.SectionName

Synopsis: Default section name

Declaration: `Property SectionName : String`

Visibility: public

Access: Read,Write

Description: `SectionName` is the section name under which the item will be saved or from which it should be read. The read/write functions should be overridden in descendents to determine a unique section name within the .ini file.

See also: TIniCollectionItem.SaveToFile (237), TIniCollectionItem.LoadFromIni (237)

17.7 TNamedIniCollection

17.7.1 Description

`TNamedIniCollection` is the collection to go with the TNamedIniCollectionItem (240) item class. it provides some functions to look for items based on the UserData (239) or based on the Name (239).

17.7.2 Method overview

Page	Property	Description
239	FindByName	Return the item based on its name
240	FindByUserData	Return the item based on its `UserData`
239	IndexOfName	Search for an item, based on its name, and return its position
239	IndexOfUserData	Search for an item based on it's `UserData` property

17.7.3 Property overview

Page	Property	Access	Description
240	NamedItems	rw	Indexed access to the `TNamedIniCollectionItem` items

17.7.4 TNamedIniCollection.IndexOfUserData

Synopsis: Search for an item based on it's `UserData` property

Declaration: `function IndexOfUserData(UserData: TObject) : Integer`

Visibility: public

Description: `IndexOfUserData` searches the list of items and returns the index of the item which has `UserData` in its UserData (239) property. If no such item exists, -1 is returned.

Note that the (linear) search starts at the last element and works it's way back to the first.

Errors: If no item exists, -1 is returned.

See also: TNamedIniCollection.IndexOfName (239), TNamedIniCollectionItem.UserData (240)

17.7.5 TNamedIniCollection.IndexOfName

Synopsis: Search for an item, based on its name, and return its position

Declaration: `function IndexOfName(const AName: String) : Integer`

Visibility: public

Description: `IndexOfName` searches the list of items and returns the index of the item which has name equal to `AName` (case insentitive). If no such item exists, -1 is returned.

Note that the (linear) search starts at the last element and works it's way back to the first.

Errors: If no item exists, -1 is returned.

See also: TNamedIniCollection.IndexOfUserData (239), TNamedIniCollectionItem.Name (241)

17.7.6 TNamedIniCollection.FindByName

Synopsis: Return the item based on its name

Declaration: `function FindByName(const AName: String) : TNamedIniCollectionItem`

Visibility: public

Description: `FindByName` returns the collection item whose name matches `AName` (case insensitive match). It calls IndexOfName (239) and returns the item at the found position. If no item is found, `Nil` is returned.

Errors: If no item is found, `Nil` is returned.

See also: TNamedIniCollection.IndexOfName (239), TNamedIniCollection.FindByUserData (240)

17.7.7 TNamedIniCollection.FindByUserData

Synopsis: Return the item based on its `UserData`

Declaration: `function FindByUserData(UserData: TObject) : TNamedIniCollectionItem`

Visibility: public

Description: `FindByName` returns the collection item whose UserData (240) property value matches the `UserData` parameter. If no item is found, `Nil` is returned.

Errors: If no item is found, `Nil` is returned.

17.7.8 TNamedIniCollection.NamedItems

Synopsis: Indexed access to the `TNamedIniCollectionItem` items

Declaration: `Property NamedItems[Index: Integer]: TNamedIniCollectionItem; default`

Visibility: public

Access: Read,Write

Description: `NamedItem` is the default property of the `TNamedIniCollection` collection. It allows indexed access to the TNamedIniCollectionItem (240) items. The index is zero based.

See also: TNamedIniCollectionItem (240)

17.8 TNamedIniCollectionItem

17.8.1 Description

`TNamedIniCollectionItem` is a TIniCollectionItem (236) descent with a published name property. The name is used as the section name when saving the item to the ini file.

17.8.2 Property overview

Page	Property	Access	Description
241	Name	rw	Name of the item
240	UserData	rw	User-defined data

17.8.3 TNamedIniCollectionItem.UserData

Synopsis: User-defined data

Declaration: `Property UserData : TObject`

Visibility: public

Access: Read,Write

Description: `UserData` can be used to associate an arbitrary object with the item - much like the `Objects` property of a `TStrings`.

17.8.4 TNamedIniCollectionItem.Name

Synopsis: Name of the item

Declaration: `Property Name : String`

Visibility: published

Access: Read,Write

Description: `Name` is the name of this item. It is also used as the section name when writing the collection item to the .ini file.

See also: TNamedIniCollectionItem.UserData (240)

Chapter 18

Reference for unit 'IniFiles'

18.1 Used units

Table 18.1: Used units by unit 'IniFiles'

Name	Page
Classes	??
contnrs	81
sysutils	??

18.2 Overview

IniFiles provides support for handling .ini files. It contains an implementation completely independent of the Windows API for handling such files. The basic (abstract) functionality is defined in TCustomInifile (242) and is implemented in TIniFile (254) and TMemIniFile (262). The API presented by these components is Delphi compatible.

18.3 TCustomIniFile

18.3.1 Description

`TCustomIniFile` implements all calls for manipulating a .ini. It does not implement any of this behaviour, the behaviour must be implemented in a descendent class like TIniFile (254) or TMemIniFile (262).

Since `TCustomIniFile` is an abstract class, it should never be created directly. Instead, one of the `TIniFile` or `TMemIniFile` classes should be created.

18.3.2 Method overview

Page	Property	Description
243	Create	Instantiate a new instance of `TCustomIniFile`.
250	DeleteKey	Delete a key from a section
244	Destroy	Remove the `TCustomIniFile` instance from memory
250	EraseSection	Clear a section
247	ReadBinaryStream	Read binary data
245	ReadBool	
246	ReadDate	Read a date value
246	ReadDateTime	Read a Date/Time value
247	ReadFloat	Read a floating point value
245	ReadInteger	Read an integer value from the file
249	ReadSection	Read the key names in a section
250	ReadSections	Read the list of sections
250	ReadSectionValues	Read names and values of a section
244	ReadString	Read a string valued key
247	ReadTime	Read a time value
244	SectionExists	Check if a section exists.
251	UpdateFile	Update the file on disk
251	ValueExists	Check if a value exists
249	WriteBinaryStream	Write binary data
246	WriteBool	Write boolean value
248	WriteDate	Write date value
248	WriteDateTime	Write date/time value
248	WriteFloat	Write a floating-point value
245	WriteInteger	Write an integer value
245	WriteString	Write a string value
249	WriteTime	Write time value

18.3.3 Property overview

Page	Property	Access	Description
252	CaseSensitive	rw	Are key and section names case sensitive
252	EscapeLineFeeds	r	Should linefeeds be escaped ?
251	FileName	r	Name of the .ini file
252	StripQuotes	rw	Should quotes be stripped from string values

18.3.4 TCustomIniFile.Create

Synopsis: Instantiate a new instance of `TCustomIniFile`.

Declaration:
```
constructor Create(const AFileName: String;AEscapeLineFeeds: Boolean)
                   ;  Virtual
```

Visibility: public

Description: `Create` creates a new instance of `TCustomIniFile` and loads it with the data from `AFileName`, if this file exists. If the `AEscapeLineFeeds` parameter is `True`, then lines which have their end-of-line markers escaped with a backslash, will be concatenated. This means that the following 2 lines

```
Description=This is a \
line with a long text
```

is equivalent to

```
Description=This is a line with a long text
```

By default, not escaping of linefeeds is performed (for Delphi compatibility)

Errors: If the file cannot be read, an exception may be raised.

See also: TCustomIniFile.Destroy (244)

18.3.5 TCustomIniFile.Destroy

Synopsis: Remove the `TCustomIniFile` instance from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up all internal structures and then calls the inherited `Destroy`.

See also: TCustomIniFile (242)

18.3.6 TCustomIniFile.SectionExists

Synopsis: Check if a section exists.

Declaration: `function SectionExists(const Section: String) : Boolean;  Virtual`

Visibility: public

Description: `SectionExists` returns `True` if a section with name `Section` exists, and contains keys. (comments are not considered keys)

See also: TCustomIniFile.ValueExists (251)

18.3.7 TCustomIniFile.ReadString

Synopsis: Read a string valued key

Declaration:
```
function ReadString(const Section: String;const Ident: String;
                    const Default: String) : String;  Virtual;  Abstract
```

Visibility: public

Description: `ReadString` reads the key `Ident` in section `Section`, and returns the value as a string. If the specified key or section do not exist, then the value in `Default` is returned. Note that if the key exists, but is empty, an empty string will be returned.

See also: TCustomIniFile.WriteString (245), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadBinaryStream (247)

18.3.8 TCustomIniFile.WriteString

Synopsis: Write a string value

Declaration:
```
procedure WriteString(const Section: String;const Ident: String;
                      const Value: String);  Virtual;  Abstract
```

Visibility: public

Description: `WriteString` writes the string `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist.

See also: TCustomIniFile.ReadString (244), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteBinaryStream (249)

18.3.9 TCustomIniFile.ReadInteger

Synopsis: Read an integer value from the file

Declaration:
```
function ReadInteger(const Section: String;const Ident: String;
                     Default: LongInt) : LongInt;  Virtual
```

Visibility: public

Description: `ReadInteger` reads the key `Ident` in section `Section`, and returns the value as an integer. If the specified key or section do not exist, then the value in `Default` is returned. If the key exists, but contains an invalid integer value, `Default` is also returned.

See also: TCustomIniFile.WriteInteger (245), TCustomIniFile.ReadString (244), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadBinaryStream (247)

18.3.10 TCustomIniFile.WriteInteger

Synopsis: Write an integer value

Declaration:
```
procedure WriteInteger(const Section: String;const Ident: String;
                       Value: LongInt);  Virtual
```

Visibility: public

Description: `WriteInteger` writes the integer `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist.

See also: TCustomIniFile.ReadInteger (245), TCustomIniFile.WriteString (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteBinaryStream (249)

18.3.11 TCustomIniFile.ReadBool

Synopsis:

Declaration:
```
function ReadBool(const Section: String;const Ident: String;
                  Default: Boolean) : Boolean;  Virtual
```

Visibility: public

Description: `ReadString` reads the key `Ident` in section `Section`, and returns the value as a boolean (valid values are 0 and 1). If the specified key or section do not exist, then the value in `Default` is returned. If the key exists, but contains an invalid integer value, `False` is also returned.

Errors:

See also: TCustomIniFile.WriteBool (246), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadString (244), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadBinaryStream (247)

18.3.12 TCustomIniFile.WriteBool

Synopsis: Write boolean value

Declaration:
```
procedure WriteBool(const Section: String;const Ident: String;
                    Value: Boolean);  Virtual
```

Visibility: public

Description: `WriteBool` writes the boolean `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist.

See also: TCustomIniFile.ReadBool (245), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteString (245), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteBinaryStream (249)

18.3.13 TCustomIniFile.ReadDate

Synopsis: Read a date value

Declaration:
```
function ReadDate(const Section: String;const Ident: String;
                  Default: TDateTime) : TDateTime;  Virtual
```

Visibility: public

Description: `ReadDate` reads the key `Ident` in section `Section`, and returns the value as a date (`TDateTime`). If the specified key or section do not exist, then the value in `Default` is returned. If the key exists, but contains an invalid date value, `Default` is also returned. The international settings of the SysUtils are taken into account when deciding if the read value is a correct date.

Errors:

See also: TCustomIniFile.WriteDate (248), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadString (244), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadBinaryStream (247)

18.3.14 TCustomIniFile.ReadDateTime

Synopsis: Read a Date/Time value

Declaration:
```
function ReadDateTime(const Section: String;const Ident: String;
                      Default: TDateTime) : TDateTime;  Virtual
```

Visibility: public

Description: `ReadDateTime` reads the key `Ident` in section `Section`, and returns the value as a date/time (`TDateTime`). If the specified key or section do not exist, then the value in `Default` is returned. If the key exists, but contains an invalid date/time value, `Default` is also returned. The international settings of the SysUtils are taken into account when deciding if the read value is a correct date/time.

See also: TCustomIniFile.WriteDateTime (248), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadString (244), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadBinaryStream (247)

18.3.15 TCustomIniFile.ReadFloat

Synopsis: Read a floating point value

Declaration:
```
function ReadFloat(const Section: String;const Ident: String;
                   Default: Double) : Double;  Virtual
```

Visibility: public

Description: `ReadFloat` reads the key `Ident` in section `Section`, and returns the value as a float (`Double`). If the specified key or section do not exist, then the value in `Default` is returned. If the key exists, but contains an invalid float value, `Default` is also returned. The international settings of the SysUtils are taken into account when deciding if the read value is a correct float.

See also: TCustomIniFile.WriteFloat (248), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadString (244), TCustomIniFile.ReadBinaryStream (247)

18.3.16 TCustomIniFile.ReadTime

Synopsis: Read a time value

Declaration:
```
function ReadTime(const Section: String;const Ident: String;
                  Default: TDateTime) : TDateTime;  Virtual
```

Visibility: public

Description: `ReadTime` reads the key `Ident` in section `Section`, and returns the value as a time (`TDateTime`). If the specified key or section do not exist, then the value in `Default` is returned. If the key exists, but contains an invalid time value, `Default` is also returned. The international settings of the SysUtils are taken into account when deciding if the read value is a correct time.

Errors:

See also: TCustomIniFile.WriteTime (249), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadString (244), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadBinaryStream (247)

18.3.17 TCustomIniFile.ReadBinaryStream

Synopsis: Read binary data

Declaration:
```
function ReadBinaryStream(const Section: String;const Name: String;
                          Value: TStream) : Integer;  Virtual
```

Visibility: public

Description: `ReadBinaryStream` reads the key `Name` in section `Section`, and returns the value in the stream `Value`. If the specified key or section do not exist, then the contents of `Value` are left untouched. The stream is not cleared prior to adding data to it.

The data is interpreted as a series of 2-byte hexadecimal values, each representing a byte in the data stream, i.e, it should always be an even number of hexadecimal characters.

See also: TCustomIniFile.WriteBinaryStream (249), TCustomIniFile.ReadInteger (245), TCustomIniFile.ReadBool (245), TCustomIniFile.ReadDate (246), TCustomIniFile.ReadDateTime (246), TCustomIniFile.ReadTime (247), TCustomIniFile.ReadFloat (247), TCustomIniFile.ReadString (244)

18.3.18 TCustomIniFile.WriteDate

Synopsis: Write date value

Declaration:
```
procedure WriteDate(const Section: String;const Ident: String;
                    Value: TDateTime);  Virtual
```

Visibility: public

Description: `WriteDate` writes the date `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist. The date is written using the internationalization settings in the SysUtils unit.

Errors:

See also: TCustomIniFile.ReadDate (246), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteString (245), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteBinaryStream (249)

18.3.19 TCustomIniFile.WriteDateTime

Synopsis: Write date/time value

Declaration:
```
procedure WriteDateTime(const Section: String;const Ident: String;
                        Value: TDateTime);  Virtual
```

Visibility: public

Description: `WriteDateTime` writes the date/time `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist. The date/time is written using the internationalization settings in the SysUtils unit.

See also: TCustomIniFile.ReadDateTime (246), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteString (245), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteBinaryStream (249)

18.3.20 TCustomIniFile.WriteFloat

Synopsis: Write a floating-point value

Declaration:
```
procedure WriteFloat(const Section: String;const Ident: String;
                     Value: Double);  Virtual
```

Visibility: public

Description: `WriteFloat` writes the time `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist. The floating point value is written using the internationalization settings in the SysUtils unit.

See also: TCustomIniFile.ReadFloat (247), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteString (245), TCustomIniFile.WriteBinaryStream (249)

18.3.21 TCustomIniFile.WriteTime

Synopsis: Write time value

Declaration:
```
procedure WriteTime(const Section: String;const Ident: String;
                    Value: TDateTime);  Virtual
```

Visibility: public

Description: `WriteTime` writes the time `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist. The time is written using the internationalization settings in the SysUtils unit.

See also: TCustomIniFile.ReadTime (247), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteString (245), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteBinaryStream (249)

18.3.22 TCustomIniFile.WriteBinaryStream

Synopsis: Write binary data

Declaration:
```
procedure WriteBinaryStream(const Section: String;const Name: String;
                            Value: TStream);  Virtual
```

Visibility: public

Description: `WriteBinaryStream` writes the binary data in `Value` with the name `Ident` to the section `Section`, overwriting any previous value that may exist there. The section will be created if it does not exist.

The binary data is encoded using a 2-byte hexadecimal value per byte in the data stream. The data stream must be seekable, so it's size can be determined. The data stream is not repositioned, it must be at the correct position.

See also: TCustomIniFile.ReadBinaryStream (247), TCustomIniFile.WriteInteger (245), TCustomIniFile.WriteBool (246), TCustomIniFile.WriteDate (248), TCustomIniFile.WriteDateTime (248), TCustomIniFile.WriteTime (249), TCustomIniFile.WriteFloat (248), TCustomIniFile.WriteString (245)

18.3.23 TCustomIniFile.ReadSection

Synopsis: Read the key names in a section

Declaration:
```
procedure ReadSection(const Section: String;Strings: TStrings);  Virtual
                      ;  Abstract
```

Visibility: public

Description: `ReadSection` will return the names of the keys in section `Section` in `Strings`, one string per key. If a non-existing section is specified, the list is cleared. To return the values of the keys as well, the ReadSectionValues (250) method should be used.

See also: TCustomIniFile.ReadSections (250), TCustomIniFile.SectionExists (244), TCustomIniFile.ReadSectionValues (250)

18.3.24 TCustomIniFile.ReadSections

Synopsis: Read the list of sections

Declaration:
```
procedure ReadSections(Strings: TStrings);  Virtual;  Abstract
```

Visibility: public

Description: `ReadSections` returns the names of existing sections in `Strings`. It also returns names of empty sections.

See also: TCustomIniFile.SectionExists (244), TCustomIniFile.ReadSectionValues (250), TCustomIniFile.ReadSection (249)

18.3.25 TCustomIniFile.ReadSectionValues

Synopsis: Read names and values of a section

Declaration:
```
procedure ReadSectionValues(const Section: String;Strings: TStrings)
                                ;  Virtual;  Abstract
```

Visibility: public

Description: `ReadSectionValues` returns the keys and their values in the section `Section` in `Strings`. They are returned as `Key=Value` strings, one per key, so the `Values` property of the stringlist can be used to read the values. To retrieve just the names of the available keys, ReadSection (249) can be used.

See also: TCustomIniFile.SectionExists (244), TCustomIniFile.ReadSections (250), TCustomIniFile.ReadSection (249)

18.3.26 TCustomIniFile.EraseSection

Synopsis: Clear a section

Declaration:
```
procedure EraseSection(const Section: String);  Virtual;  Abstract
```

Visibility: public

Description: `EraseSection` deletes all values from the section named `Section` and removes the section from the ini file. If the section didn't exist prior to a call to `EraseSection`, nothing happens.

See also: TCustomIniFile.SectionExists (244), TCustomIniFile.ReadSections (250), TCustomIniFile.DeleteKey (250)

18.3.27 TCustomIniFile.DeleteKey

Synopsis: Delete a key from a section

Declaration:
```
procedure DeleteKey(const Section: String;const Ident: String);  Virtual
                        ;  Abstract
```

Visibility: public

Description: `DeleteKey` deletes the key `Ident` from section `Section`. If the key or section didn't exist prior to the `DeleteKey` call, nothing happens.

See also: TCustomIniFile.EraseSection (250)

18.3.28 TCustomIniFile.UpdateFile

Synopsis: Update the file on disk

Declaration:
```
procedure UpdateFile;  Virtual;  Abstract
```

Visibility: public

Description: `UpdateFile` writes the in-memory image of the ini-file to disk. To speed up operation of the inifile class, the whole ini-file is read into memory when the class is created, and all operations are performed in-memory. If `CacheUpdates` is set to `True`, any changes to the inifile are only in memory, until they are committed to disk with a call to `UpdateFile`. If `CacheUpdates` is set to `False`, then all operations which cause a change in the .ini file will immediatly be committed to disk with a call to `UpdateFile`. Since the whole file is written to disk, this may have serious impact on performance.

See also: TIniFile.CacheUpdates (257)

18.3.29 TCustomIniFile.ValueExists

Synopsis: Check if a value exists

Declaration:
```
function ValueExists(const Section: String;const Ident: String)
                          : Boolean;  Virtual
```

Visibility: public

Description: `ValueExists` checks whether the key `Ident` exists in section `Section`. It returns `True` if a key was found, or `False` if not. The key may be empty.

See also: TCustomIniFile.SectionExists (244)

18.3.30 TCustomIniFile.FileName

Synopsis: Name of the .ini file

Declaration:
```
Property FileName : String
```

Visibility: public

Access: Read

Description: `FileName` is the name of the ini file on disk. It should be specified when the `TCustomIniFile` instance is created. Contrary to the Delphi implementation, if no path component is present in the filename, the filename is not searched in the windows directory.

See also: TCustomIniFile.Create (243)

18.3.31 TCustomIniFile.EscapeLineFeeds

Synopsis: Should linefeeds be escaped ?

Declaration: `Property EscapeLineFeeds : Boolean`

Visibility: public

Access: Read

Description: `EscapeLineFeeds` determines whether escaping of linefeeds is enabled: For a description of this feature, see Create (243), as the value of this property must be specified when the `TCustomIniFile` instance is created.

By default, `EscapeLineFeeds` is `False`.

See also: TCustomIniFile.Create (243), TCustomIniFile.CaseSensitive (252)

18.3.32 TCustomIniFile.CaseSensitive

Synopsis: Are key and section names case sensitive

Declaration: `Property CaseSensitive : Boolean`

Visibility: public

Access: Read,Write

Description: `CaseSensitive` determines whether searches for sections and keys are performed case-sensitive or not. By default, they are not case sensitive.

See also: TCustomIniFile.EscapeLineFeeds (252)

18.3.33 TCustomIniFile.StripQuotes

Synopsis: Should quotes be stripped from string values

Declaration: `Property StripQuotes : Boolean`

Visibility: public

Access: Read,Write

Description: `StripQuotes` determines whether quotes around string values are stripped from the value when reading the values from file. By default, quotes are not stripped (this is Delphi and Windows compatible).

18.4 THashedStringList

18.4.1 Description

`THashedStringList` is a TStringList (**??**) descendent which creates has values for the strings and names (in the case of a name-value pair) stored in it. The IndexOf (253) and IndexOfName (253) functions make use of these hash values to quicklier locate a value.

18.4.2 Method overview

Page	Property	Description
253	Create	Instantiates a new instance of `THashedStringList`
253	Destroy	Clean up instance
253	IndexOf	Returns the index of a string in the list of strings
253	IndexOfName	Return the index of a name in the list of name=value pairs

18.4.3 THashedStringList.Create

Synopsis: Instantiates a new instance of `THashedStringList`

Declaration: `constructor Create`

Visibility: public

Description: `Create` calls the inherited `Create`, and then instantiates the hash tables.

Errors: If no enough memory is available, an exception may be raised.

See also: THashedStringList.Destroy (253)

18.4.4 THashedStringList.Destroy

Synopsis: Clean up instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the hash tables and then calls the inherited `Destroy`.

See also: THashedStringList.Create (253)

18.4.5 THashedStringList.IndexOf

Synopsis: Returns the index of a string in the list of strings

Declaration: `function IndexOf(const S: String) : Integer;  Override`

Visibility: public

Description: `IndexOf` overrides the TStringList.IndexOf (242) method and uses the hash values to look for the location of `S`.

See also: TStringList.IndexOf (242), THashedStringList.IndexOfName (253)

18.4.6 THashedStringList.IndexOfName

Synopsis: Return the index of a name in the list of name=value pairs

Declaration: `function IndexOfName(const Name: String) : Integer;  Override`

Visibility: public

Description: `IndexOfName` overrides the TStrings.IndexOfName (242) method and uses the hash values of the names to look for the location of `Name`.

See also: TStrings.IndexOfName (242), THashedStringList.IndexOf (253)

18.5 TIniFile

18.5.1 Description

`TIniFile` is an implementation of TCustomIniFile (242) which does the same as TMemIniFile (262), namely it reads the whole file into memory. Unlike `TMemIniFile` it does not cache updates in memory, but immediatly writes any changes to disk.

`TIniFile` introduces no new methods, it just implements the abstract methods introduced in `TCustomIniFile`

18.5.2 Method overview

Page	Property	Description
254	Create	Create a new instance of `TIniFile`
256	DeleteKey	Delete key
254	Destroy	Remove the `TIniFile` instance from memory
256	EraseSection	
255	ReadSection	Read the key names in a section
255	ReadSectionRaw	Read raw section
256	ReadSections	Read section names
256	ReadSectionValues	
255	ReadString	Read a string
257	UpdateFile	Update the file on disk
255	WriteString	Write string to file

18.5.3 Property overview

Page	Property	Access	Description
257	CacheUpdates	rw	Should changes be kept in memory
257	Stream	r	Stream from which ini file was read

18.5.4 TIniFile.Create

Synopsis: Create a new instance of `TIniFile`

Declaration:
```
constructor Create(const AFileName: String;AEscapeLineFeeds: Boolean)
                    ;  Override
constructor Create(AStream: TStream;AEscapeLineFeeds: Boolean)
```

Visibility: public

Description: `Create` creates a new instance of `TIniFile` and initializes the class by reading the file from disk if the filename `AFileName` is specified, or from stream in case `AStream` is specified. It also sets most variables to their initial values, i.e. `AEscapeLineFeeds` is saved prior to reading the file, and `Cacheupdates` is set to `False`.

See also: TCustomIniFile (242), TMemIniFile (262)

18.5.5 TIniFile.Destroy

Synopsis: Remove the `TIniFile` instance from memory

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` writes any pending changes to disk, and cleans up the `TIniFile` structures, and then calls the inherited `Destroy`, effectively removing the instance from memory.

Errors: If an error happens when the file is written to disk, an exception will be raised.

See also: TCustomIniFile.UpdateFile (251), TIniFile.CacheUpdates (257)

18.5.6 TIniFile.ReadString

Synopsis: Read a string

Declaration:
```
function ReadString(const Section: String;const Ident: String;
                    const Default: String) : String;  Override
```

Visibility: public

Description: `ReadString` implements the TCustomIniFile.ReadString (244) abstract method by looking at the in-memory copy of the ini file and returning the string found there.

See also: TCustomIniFile.ReadString (244)

18.5.7 TIniFile.WriteString

Synopsis: Write string to file

Declaration:
```
procedure WriteString(const Section: String;const Ident: String;
                      const Value: String);  Override
```

Visibility: public

Description: `WriteString` implements the TCustomIniFile.WriteString (245) abstract method by writing the string to the in-memory copy of the ini file. If CacheUpdates (257) property is `False`, then the whole file is immediatly written to disk as well.

Errors: If an error happens when the file is written to disk, an exception will be raised.

18.5.8 TIniFile.ReadSection

Synopsis: Read the key names in a section

Declaration:
```
procedure ReadSection(const Section: String;Strings: TStrings)
                      ;  Override
```

Visibility: public

Description: `ReadSection` reads the key names from `Section` into `Strings`, taking the in-memory copy of the ini file. This is the implementation for the abstract TCustomIniFile.ReadSection (249)

See also: TCustomIniFile.ReadSection (249), TIniFile.ReadSectionRaw (255)

18.5.9 TIniFile.ReadSectionRaw

Synopsis: Read raw section

Declaration:
```
procedure ReadSectionRaw(const Section: String;Strings: TStrings)
```

Visibility: public

Description: `ReadSectionRaw` returns the contents of the section `Section` as it is: this includes the comments in the section. (these are also stored in memory)

See also: TIniFile.ReadSection (255), TCustomIniFile.ReadSection (249)

18.5.10 TIniFile.ReadSections

Synopsis: Read section names

Declaration:
```
procedure ReadSections(Strings: TStrings);  Override
```

Visibility: public

Description: `ReadSections` is the implementation of TCustomIniFile.ReadSections (250). It operates on the in-memory copy of the inifile, and places all section names in `Strings`.

See also: TIniFile.ReadSection (255), TCustomIniFile.ReadSections (250), TIniFile.ReadSectionValues (256)

18.5.11 TIniFile.ReadSectionValues

Synopsis:

Declaration:
```
procedure ReadSectionValues(const Section: String;Strings: TStrings)
                                ;  Override
```

Visibility: public

Description: `ReadSectionValues` is the implementation of TCustomIniFile.ReadSectionValues (250). It operates on the in-memory copy of the inifile, and places all key names from `Section` together with their values in `Strings`.

See also: TIniFile.ReadSection (255), TCustomIniFile.ReadSectionValues (250), TIniFile.ReadSections (256)

18.5.12 TIniFile.EraseSection

Synopsis:

Declaration:
```
procedure EraseSection(const Section: String);  Override
```

Visibility: public

Description: `Erasesection` deletes the section `Section` from memory, if CacheUpdates (257) is `False`, then the file is immediatly updated on disk. This method is the implementation of the abstract TCustomIniFile.EraseSection (250) method.

See also: TCustomIniFile.EraseSection (250), TIniFile.ReadSection (255), TIniFile.ReadSections (256)

18.5.13 TIniFile.DeleteKey

Synopsis: Delete key

Declaration:
```
procedure DeleteKey(const Section: String;const Ident: String)
                        ;  Override
```

Visibility: public

Description: `DeleteKey` deletes the `Ident` from the section `Section`. This operation is performed on the in-memory copy of the ini file. if CacheUpdates (257) is `False`, then the file is immediatly updated on disk.

See also: TIniFile.CacheUpdates (257)

18.5.14 TIniFile.UpdateFile

Synopsis: Update the file on disk

Declaration: `procedure UpdateFile;  Override`

Visibility: public

Description: `UpdateFile` writes the in-memory data for the ini file to disk. The whole file is written. If the ini file was instantiated from a stream, then the stream is updated. Note that the stream must be seekable for this to work correctly. The ini file is marked as 'clean' after a call to `UpdateFile` (i.e. not in need of writing to disk).

Errors: If an error occurs when writing to stream or disk, an exception may be raised.

See also: TIniFile.CacheUpdates (257)

18.5.15 TIniFile.Stream

Synopsis: Stream from which ini file was read

Declaration: `Property Stream : TStream`

Visibility: public

Access: Read

Description: `Stream` is the stream which was used to create the `IniFile`. The UpdateFile (257) method will use this stream to write changes to.

See also: TIniFile.Create (254), TIniFile.UpdateFile (257)

18.5.16 TIniFile.CacheUpdates

Synopsis: Should changes be kept in memory

Declaration: `Property CacheUpdates : Boolean`

Visibility: public

Access: Read,Write

Description: `CacheUpdates` determines how to deal with changes to the ini-file data: if set to `True` then changes are kept in memory till the file is written to disk with a call to UpdateFile (257). If it is set to `False` then each call that changes the data of the ini-file will result in a call to `UpdateFile`. This is the default behaviour, but it may aversely affect performance.

See also: TIniFile.UpdateFile (257)

18.6 TIniFileKey

18.6.1 Description

`TIniFileKey` is used to keep the key/value pairs in the ini file in memory. It is an internal structure, used internally by the TIniFile (254) class.

18.6.2 Method overview

Page	Property	Description
258	Create	Create a new instance of `TIniFileKey`

18.6.3 Property overview

Page	Property	Access	Description
258	Ident	rw	Key name
258	Value	rw	Key value

18.6.4 TIniFileKey.Create

Synopsis: Create a new instance of `TIniFileKey`

Declaration: `constructor Create(AIdent: String;AValue: String)`

Visibility: public

Description: `Create` instantiates a new instance of `TIniFileKey` on the heap. It fills Ident (258) with `AIdent` and Value (258) with `AValue`.

See also: TIniFileKey.Ident (258), TIniFileKey.Value (258)

18.6.5 TIniFileKey.Ident

Synopsis: Key name

Declaration: `Property Ident : String`

Visibility: public

Access: Read,Write

Description: `Ident` is the key value part of the key/value pair.

See also: TIniFileKey.Value (258)

18.6.6 TIniFileKey.Value

Synopsis: Key value

Declaration: `Property Value : String`

Visibility: public

Access: Read,Write

Description: `Value` is the value part of the key/value pair.

See also: TIniFileKey.Ident (258)

18.7 TIniFileKeyList

18.7.1 Description

`TIniFileKeyList` maintains a list of TIniFileKey (258) instances on behalf of the TIniFileSection (260) class. It stores they keys of one section of the .ini files.

18.7.2 Method overview

Page	Property	Description
259	Clear	Clear the list
259	Destroy	Free the instance

18.7.3 Property overview

Page	Property	Access	Description
259	Items	r	Indexed access to `TIniFileKey` items in the list

18.7.4 TIniFileKeyList.Destroy

Synopsis: Free the instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` clears up the list using Clear (259) and then calls the inherited destroy.

See also: TIniFileKeyList.Clear (259)

18.7.5 TIniFileKeyList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear;  Override`

Visibility: public

Description: `Clear` removes all TIniFileKey (258) instances from the list, and frees the instances.

See also: TIniFileKey (258)

18.7.6 TIniFileKeyList.Items

Synopsis: Indexed access to `TIniFileKey` items in the list

Declaration: `Property Items[Index: Integer]: TIniFileKey; default`

Visibility: public

Access: Read

Description: `Items` provides indexed access to the TIniFileKey (258) items in the list. The index is zero-based and runs from 0 to `Count-1`.

See also: TIniFileKey (258)

18.8 TIniFileSection

18.8.1 Description

`TIniFileSection` is a class which represents a section in the .ini, and is used internally by the TIniFile (254) class (one instance of `TIniFileSection` is created for each section in the file by the TIniFileSectionList (261) list). The name of the section is stored in the Name (261) property, and the key/value pairs in this section are available in the KeyList (261) property.

18.8.2 Method overview

Page	Property	Description
260	Create	Create a new section object
260	Destroy	Free the section object from memory
260	Empty	Is the section empty

18.8.3 Property overview

Page	Property	Access	Description
261	KeyList	r	List of key/value pairs in this section
261	Name	r	Name of the section

18.8.4 TIniFileSection.Empty

Synopsis: Is the section empty

Declaration:
```
function Empty : Boolean
```

Visibility: public

Description: `Empty` returns `True` if the section contains no key values (even if they are empty). It may contain comments.

18.8.5 TIniFileSection.Create

Synopsis: Create a new section object

Declaration:
```
constructor Create(AName: String)
```

Visibility: public

Description: `Create` instantiates a new `TIniFileSection` class, and sets the name to `AName`. It allocates a TIniFileKeyList (259) instance to keep all the key/value pairs for this section.

See also: TIniFileKeyList (259)

18.8.6 TIniFileSection.Destroy

Synopsis: Free the section object from memory

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` cleans up the key list, and then calls the inherited `Destroy`, removing the `TIniFileSection` instance from memory.

See also: TIniFileSection.Create (260), TIniFileKeyList (259)

18.8.7 TIniFileSection.Name

Synopsis: Name of the section

Declaration: `Property Name : String`

Visibility: public

Access: Read

Description: `Name` is the name of the section in the file.

See also: TIniFileSection.KeyList (261)

18.8.8 TIniFileSection.KeyList

Synopsis: List of key/value pairs in this section

Declaration: `Property KeyList : TIniFileKeyList`

Visibility: public

Access: Read

Description: `KeyList` is the TIniFileKeyList (259) instance that is used by the `TIniFileSection` to keep the key/value pairs of the section.

See also: TIniFileSection.Name (261), TIniFileKeyList (259)

18.9 TIniFileSectionList

18.9.1 Description

`TIniFileSectionList` maintains a list of TIniFileSection (260) instances, one for each section in an .ini file. `TIniFileSectionList` is used internally by the TIniFile (254) class to represent the sections in the file.

18.9.2 Method overview

Page	Property	Description
262	Clear	Clear the list
262	Destroy	Free the object from memory

18.9.3 Property overview

Page	Property	Access	Description
262	Items	r	Indexed access to all the section objects in the list

18.9.4 TIniFileSectionList.Destroy

Synopsis: Free the object from memory

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` calls Clear (262) to clear the section list and the calls the inherited `Destroy`

See also: TIniFileSectionList.Clear (262)

18.9.5 TIniFileSectionList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear; Override`

Visibility: public

Description: `Clear` removes all TIniFileSection (260) items from the list, and frees the items it removes from the list.

See also: TIniFileSection (260), TIniFileSectionList.Items (262)

18.9.6 TIniFileSectionList.Items

Synopsis: Indexed access to all the section objects in the list

Declaration: `Property Items[Index: Integer]: TIniFileSection; default`

Visibility: public

Access: Read

Description: `Items` provides indexed access to all the section objects in the list. `Index` should run from 0 to `Count-1`.

See also: TIniFileSection (260), TIniFileSectionList.Clear (262)

18.10 TMemIniFile

18.10.1 Description

`TMemIniFile` is a simple descendent of TIniFile (254) which introduces some extra methods to be compatible to the Delphi implementation of `TMemIniFile`. The FPC implementation of `TIniFile` is implemented as a `TMemIniFile`, except that `TIniFile` does not cache its updates, and `TMemIniFile` does.

18.10.2 Method overview

Page	Property	Description
263	Clear	Clear the data
263	Create	Create a new instance of `TMemIniFile`
263	GetStrings	Get contents of ini file as stringlist
263	Rename	Rename the ini file
264	SetStrings	Set data from a stringlist

18.10.3 TMemIniFile.Create

Synopsis: Create a new instance of `TMemIniFile`

Declaration:
```
constructor Create(const AFileName: String;AEscapeLineFeeds: Boolean)
                        ;  Override
```

Visibility: public

Description: `Create` simply calls the inherited Create (254), and sets the CacheUpdates (257) to `True` so updates will be kept in memory till they are explicitly written to disk.

See also: TIniFile.Create (254), TIniFile.CacheUpdates (257)

18.10.4 TMemIniFile.Clear

Synopsis: Clear the data

Declaration:
```
procedure Clear
```

Visibility: public

Description: `Clear` removes all sections and key/value pairs from memory. If CacheUpdates (257) is set to `False` then the file on disk will immediatly be emptied.

See also: TMemIniFile.SetStrings (264), TMemIniFile.GetStrings (263)

18.10.5 TMemIniFile.GetStrings

Synopsis: Get contents of ini file as stringlist

Declaration:
```
procedure GetStrings(List: TStrings)
```

Visibility: public

Description: `GetStrings` returns the whole contents of the ini file in a single stringlist, `List`. This includes comments and empty sections.

The `GetStrings` call can be used to get data for a call to SetStrings (264), which can be used to copy data between 2 in-memory ini files.

See also: TMemIniFile.SetStrings (264), TMemIniFile.Clear (263)

18.10.6 TMemIniFile.Rename

Synopsis: Rename the ini file

Declaration:
```
procedure Rename(const AFileName: String;Reload: Boolean)
```

Visibility: public

Description: `Rename` will rename the ini file with the new name `AFileName`. If `Reload` is `True` then the in-memory contents will be cleared and replaced with the contents found in `AFileName`, if it exists. If `Reload` is `False`, the next call to `UpdateFile` will replace the contents of `AFileName` with the in-memory data.

See also: TIniFile.UpdateFile (257)

18.10.7 TMemIniFile.SetStrings

Synopsis: Set data from a stringlist

Declaration: `procedure SetStrings(List: TStrings)`

Visibility: public

Description: `SetStrings` sets the in-memory data from the `List` stringlist. The data is first cleared.

The `SetStrings` call can be used to set the data of the ini file to a list of strings obtained with GetStrings (263). The two calls combined can be used to copy data between 2 in-memory ini files.

See also: TMemIniFile.GetStrings (263), TMemIniFile.Clear (263)

Chapter 19

Reference for unit 'iostream'

19.1 Used units

Table 19.1: Used units by unit 'iostream'

Name	Page
Classes	**??**

19.2 Overview

The iostream implements a descendent of THandleStream (**??**) streams that can be used to read from standard input and write to standard output and standard diagnostic output (`stderr`).

19.3 Constants, types and variables

19.3.1 Types

```
TIOSType = (iosInput,iosOutPut,iosError)
```

Table 19.2: Enumeration values for type TIOSType

Value	Explanation
iosError	The stream can be used to write to standard diagnostic output
iosInput	The stream can be used to read from standard input
iosOutPut	The stream can be used to write to standard output

`TIOSType` is passed to the Create (266) constructor of TIOStream (266), it determines what kind of stream is created.

19.4 EIOStreamError

19.4.1 Description

Error thrown in case of an invalid operation on a TIOStream (266).

19.5 TIOStream

19.5.1 Description

`TIOStream` can be used to create a stream which reads from or writes to the standard input, output or stderr file descriptors. It is a descendent of `THandleStream`. The type of stream that is created is determined by the TIOSType (265) argument to the constructor. The handle of the standard input, output or stderr file descriptors is determined automatically.

The `TIOStream` keeps an internal `Position`, and attempts to provide minimal Seek (267) behaviour based on this position.

19.5.2 Method overview

Page	Property	Description
266	Create	Construct a new instance of TIOStream (266)
266	Read	Read data from the stream.
267	Seek	Set the stream position
267	SetSize	Set the size of the stream
267	Write	Write data to the stream

19.5.3 TIOStream.Create

Synopsis: Construct a new instance of TIOStream (266)

Declaration: `constructor Create(aIOSType: TIOSType)`

Visibility: public

Description: `Create` creates a new instance of TIOStream (266), which can subsequently be used

Errors: No checking is performed to see whether the requested file descriptor is actually open for reading/writing. In that case, subsequent calls to `Read` or `Write` or `seek` will fail.

See also: TIOStream.Read (266), TIOStream.Write (267)

19.5.4 TIOStream.Read

Synopsis: Read data from the stream.

Declaration: `function Read(var Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` checks first whether the type of the stream allows reading (type is `iosInput`). If not, it raises a EIOStreamError (266) exception. If the stream can be read, it calls the inherited `Read` to actually read the data.

Errors: An `EIOStreamError` exception is raised if the stream does not allow reading.

See also: TIOSType (265), TIOStream.Write (267)

19.5.5 TIOStream.Write

Synopsis: Write data to the stream

Declaration: `function Write(const Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Write` checks first whether the type of the stream allows writing (type is `iosOutput` or `iosError`). If not, it raises a EIOStreamError (266) exception. If the stream can be written to, it calls the inherited `Write` to actually read the data.

Errors: An `EIOStreamError` exception is raised if the stream does not allow writing.

See also: TIOSType (265), TIOStream.Read (266)

19.5.6 TIOStream.SetSize

Synopsis: Set the size of the stream

Declaration: `procedure SetSize(NewSize: LongInt);  Override`

Visibility: public

Description: `SetSize` overrides the standard `SetSize` implementation. It always raises an exception, because the standard input, output and stderr files have no size.

Errors: An `EIOStreamError` exception is raised when this method is called.

See also: EIOStreamError (266)

19.5.7 TIOStream.Seek

Synopsis: Set the stream position

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` overrides the standard `Seek` implementation. Normally, standard input, output and stderr are not seekable. The `TIOStream` stream tries to provide seek capabilities for the following limited number of cases:

Origin=soFromBeginning If `Offset` is larger than the current position, then the remaining bytes are skipped by reading them from the stream and discarding them, if the stream is of type `iosInput`.

Origin=soFromCurrent If `Offset` is zero, the current position is returned. If it is positive, then `Offset` bytes are skipped by reading them from the stream and discarding them, if the stream is of type `iosInput`.

All other cases will result in a `EIOStreamError` exception.

Errors: An EIOStreamError (266) exception is raised if the stream does not allow the requested seek operation.

See also: EIOStreamError (266)

Chapter 20

Reference for unit 'libtar'

20.1 Used units

Table 20.1: Used units by unit 'libtar'

Name	Page
BaseUnix	268
Classes	??
sysutils	??
Unix	268
UnixType	268
Windows	268

20.2 Overview

The libtar units provides 2 classes to read and write .tar archives: TTarArchive (272) class can be used to read a tar file, and the TTarWriter (274) class can be used to write a tar file. The unit was implemented originally by Stefan Heymann.

20.3 Constants, types and variables

20.3.1 Constants

```
ALL_PERMISSIONS = [tpReadByOwner,tpWriteByOwner,tpExecuteByOwner,tpReadByGroup,tpWri
```

ALL_PERMISSIONS is a set constant containing all possible permissions (read/write/execute, for all groups of users) for an archive entry.

```
EXECUTE_PERMISSIONS = [tpExecuteByOwner,tpExecuteByGroup,tpExecuteByOther]
```

WRITE_PERMISSIONS is a set constant containing all possible execute permissions set for an archive entry.

```
FILETYPE_NAME : Array[TFileType] of String = ('Regular','Link','Symbolic Link','Char
```

FILETYPE_NAME can be used to get a textual description for each of the possible entry file types.

```
READ_PERMISSIONS = [tpReadByOwner,tpReadByGroup,tpReadByOther]
```

READ_PERMISSIONS is a set constant containing all possible read permissions set for an archive entry.

```
WRITE_PERMISSIONS = [tpWriteByOwner,tpWriteByGroup,tpWriteByOther]
```

WRITE_PERMISSIONS is a set constant containing all possible write permissions set for an archive entry.

20.3.2 Types

```
TFileType = (ftNormal,ftLink,ftSymbolicLink,ftCharacter,ftBlock,
             ftDirectory,ftFifo,ftContiguous,ftDumpDir,ftMultiVolume,
             ftVolumeHeader)
```

Table 20.2: Enumeration values for type TFileType

Value	Explanation
ftBlock	Block device file
ftCharacter	Character device file
ftContiguous	Contiguous file
ftDirectory	Directory
ftDumpDir	List of files
ftFifo	FIFO file
ftLink	Hard link
ftMultiVolume	Multi-volume file part
ftNormal	Normal file
ftSymbolicLink	Symbolic link
ftVolumeHeader	Volume header, can appear only as first entry in the archive

TFileType describes the file type of a file in the archive. It is used in the FileType field of the TTarDirRec (270) record.

```
TTarDirRec = record
  Name : String;
  Size : Int64;
  DateTime : TDateTime;
  Permissions : TTarPermissions;
  FileType : TFileType;
  LinkName : String;
  UID : Integer;
  GID : Integer;
  UserName : String;
  GroupName : String;
  ChecksumOK : Boolean;
```

```
  Mode : TTarModes;
  Magic : String;
  MajorDevNo : Integer;
  MinorDevNo : Integer;
  FilePos : Int64;
end
```

TTarDirRec describes an entry in the tar archive. It is similar to a directory entry as in TSearchRec (**??**), and is returned by the TTarArchive.FindNext (273) call.

```
TTarMode = (tmSetUid,tmSetGid,tmSaveText)
```

Table 20.3: Enumeration values for type TTarMode

Value	Explanation
tmSaveText	Bit $200 is set
tmSetGid	File has SetGID bit set
tmSetUid	File has SetUID bit set.

TTarMode describes extra file modes. It is used in the Mode field of the TTarDirRec (270) record.

```
TTarModes= Set of (tmSaveText,tmSetGid,tmSetUid)
```

TTarModes denotes the full set of permission bits for the file in the field Mode field of the TTarDirRec (270) record.

```
TTarPermission = (tpReadByOwner,tpWriteByOwner,tpExecuteByOwner,
                  tpReadByGroup,tpWriteByGroup,tpExecuteByGroup,
                  tpReadByOther,tpWriteByOther,tpExecuteByOther)
```

Table 20.4: Enumeration values for type TTarPermission

Value	Explanation
tpExecuteByGroup	Group can execute the file
tpExecuteByOther	Other people can execute the file
tpExecuteByOwner	Owner can execute the file
tpReadByGroup	Group can read the file
tpReadByOther	Other people can read the file.
tpReadByOwner	Owner can read the file
tpWriteByGroup	Group can write the file
tpWriteByOther	Other people can write the file
tpWriteByOwner	Owner can write the file

TTarPermission denotes part of a files permission as it it stored in the .tar archive. Each of these enumerated constants correspond with one of the permission bits from a unix file permission.

```
TTarPermissions= Set of (tpExecuteByGroup,tpExecuteByOther,
                         tpExecuteByOwner,tpReadByGroup,tpReadByOther,
                         tpReadByOwner,tpWriteByGroup,tpWriteByOther,
                         tpWriteByOwner)
```

TTarPermissions describes the complete set of permissions that a file has. It is used in the Permissions field of the TTarDirRec (270) record.

20.4 Procedures and functions

20.4.1 ClearDirRec

Synopsis: Initialize tar archive entry

Declaration:
```
procedure ClearDirRec(var DirRec: TTarDirRec)
```

Visibility: default

Description: ClearDirRec clears the DirRec entry, it basically zeroes out all fields.

See also: TTarDirRec (270)

20.4.2 ConvertFilename

Synopsis: Convert filename to archive format

Declaration:
```
function ConvertFilename(Filename: String) : String
```

Visibility: default

Description: ConvertFileName converts the file name FileName to a format allowed by the tar archive. Basically, it converts directory specifiers to forward slashes.

20.4.3 FileTimeGMT

Synopsis: Extract filetime

Declaration:
```
function FileTimeGMT(FileName: String) : TDateTime;  Overload
function FileTimeGMT(SearchRec: TSearchRec) : TDateTime;  Overload
```

Visibility: default

Description: FileTimeGMT returns the timestamp of a filename (FileName must exist) or a search rec (TSearchRec) to a GMT representation that can be used in a tar entry.

See also: TTarDirRec (270)

20.4.4 PermissionString

Synopsis: Convert a set of permissions to a string

Declaration:
```
function PermissionString(Permissions: TTarPermissions) : String
```

Visibility: default

Description: PermissionString can be used to convert a set of Permissions to a string in the same format as used by the unix 'ls' command.

See also: TTarPermissions (271)

20.5 TTarArchive

20.5.1 Description

`TTarArchive` is the class used to read and examine .tar archives. It can be constructed from a stream or from a filename. Creating an instance will not perform any operation on the stream yet.

20.5.2 Method overview

Page	Property	Description
272	Create	Create a new instance of the archive
272	Destroy	Destroy `TTarArchive` instance
273	FindNext	Find next archive entry
273	GetFilePos	Return current archive position
273	ReadFile	Read a file from the archive
272	Reset	Reset archive
274	SetFilePos	Set position in archive

20.5.3 TTarArchive.Create

Synopsis: Create a new instance of the archive

Declaration:
```
constructor Create(Stream: TStream);  Overload
constructor Create(Filename: String;FileMode: Word);  Overload
```

Visibility: public

Description: `Create` can be used to create a new instance of `TTarArchive` using either a `Stream`TStream (**??**) descendent or using a name of a file to open: `FileName`. In case of the filename, an open mode can be specified.

Errors: In case a filename is specified and the file cannot be opened, an exception will occur.

See also: TTarArchive.FindNext (273)

20.5.4 TTarArchive.Destroy

Synopsis: Destroy `TTarArchive` instance

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` closes the archive stream (if it created a stream) and cleans up the `TTarArchive` instance.

See also: TTarArchive.Create (272)

20.5.5 TTarArchive.Reset

Synopsis: Reset archive

Declaration:
```
procedure Reset
```

Visibility: public

Description: `Reset` sets the archive file position on the beginning of the archive.

See also: TTarArchive.Create (272)

20.5.6 TTarArchive.FindNext

Synopsis: Find next archive entry

Declaration:
```
function FindNext(var DirRec: TTarDirRec) : Boolean
```

Visibility: public

Description: `FindNext` positions the file pointer on the next archive entry, and returns all information about the entry in `DirRec`. It returns `True` if the operation was succeful, or `False` if not (for instance, when the end of the archive was reached).

Errors: In case there are no more entries, `False` is returned.

See also: TTarArchive.ReadFile (273)

20.5.7 TTarArchive.ReadFile

Synopsis: Read a file from the archive

Declaration:
```
procedure ReadFile(Buffer: POINTER);  Overload
procedure ReadFile(Stream: TStream);  Overload
procedure ReadFile(Filename: String);  Overload
function ReadFile : String;  Overload
```

Visibility: public

Description: `ReadFile` can be used to read the current file in the archive. It can be called after the archive was succesfully positioned on an entry in the archive. The file can be read in various ways:

- directly in a memory `buffer`. No checks are performed to see whether the buffer points to enough memory.
- It can be copied to a `Stream`.
- It can be copied to a file with name `FileName`.
- The file content can be copied to a string

Errors: An exception may occur if the buffer is not large enough, or when the file specified in `filename` cannot be opened.

20.5.8 TTarArchive.GetFilePos

Synopsis: Return current archive position

Declaration:
```
procedure GetFilePos(var Current: Int64;var Size: Int64)
```

Visibility: public

Description: `GetFilePos` returns the position in the tar archive in `Current` and the complete archive size in `Size`.

See also: TTarArchive.SetFilePos (274), TTarArchive.Reset (272)

20.5.9 TTarArchive.SetFilePos

Synopsis: Set position in archive

Declaration:
```
procedure SetFilePos(NewPos: Int64)
```

Visibility: public

Description: `SetFilePos` can be used to set the absolute position in the tar archive.

See also: TTarArchive.Reset (272), TTarArchive.GetFilePos (273)

20.6 TTarWriter

20.6.1 Description

`TTarWriter` can be used to create .tar archives. It can be created using a filename, in which case the archive will be written to the filename, or it can be created using a stream, in which case the archive will be written to the stream - for instance a compression stream.

20.6.2 Method overview

Page	Property	Description
276	AddDir	Add directory to archive
275	AddFile	Add a file to the archive
277	AddLink	Add hard link to archive
275	AddStream	Add stream contents to archive.
276	AddString	Add string as file data
276	AddSymbolicLink	Add a symbolic link to the archive
277	AddVolumeHeader	Add volume header entry
274	Create	Create a new archive
275	Destroy	Close archive and clean up `TTarWriter`
277	Finalize	Finalize the archive

20.6.3 Property overview

Page	Property	Access	Description
278	GID	rw	Archive entry group ID
278	GroupName	rw	Archive entry group name
279	Magic	rw	Archive entry Magic constant
279	Mode	rw	Archive entry mode
277	Permissions	rw	Archive entry permissions
278	UID	rw	Archive entry user ID
278	UserName	rw	Archive entry user name

20.6.4 TTarWriter.Create

Synopsis: Create a new archive

Declaration:
```
constructor Create(TargetStream: TStream);  Overload
constructor Create(TargetFilename: String;Mode: Integer);  Overload
```

Visibility: public

Description: `Create` creates a new `TTarWriter` instance. This will start a new .tar archive. The archive will be written to the `TargetStream` stream or to a file with name `TargetFileName`, which will be opened with filemode `Mode`.

Errors: In case `TargetFileName` cannot be opened, an exception will be raised.

See also: TTarWriter.Destroy (275)

20.6.5 TTarWriter.Destroy

Synopsis: Close archive and clean up `TTarWriter`

Declaration:
```
destructor Destroy;  Override
```

Visibility: public

Description: `Destroy` will close the archive (i.e. it writes the end-of-archive marker, if it was not yet written), and then frees the `TTarWriter` instance.

See also: TTarWriter.Finalize (277)

20.6.6 TTarWriter.AddFile

Synopsis: Add a file to the archive

Declaration:
```
procedure AddFile(Filename: String;TarFilename: String)
```

Visibility: public

Description: `AddFile` adds a file to the archive: the contents is read from `FileName`. Optionally, an alternative filename can be specified in `TarFileName`. This name should contain only forward slash path separators. If it is not specified, the name will be computed from `FileName`.

The archive entry is written with the current owner data and permissions.

Errors: If `FileName` cannot be opened, an exception will be raised.

See also: TTarWriter.AddStream (275), TTarWriter.AddString (276), TTarWriter.AddLink (277), TTarWriter.AddSymbolicLink (276), TTarWriter.AddDir (276), TTarWriter.AddVolumeHeader (277)

20.6.7 TTarWriter.AddStream

Synopsis: Add stream contents to archive.

Declaration:
```
procedure AddStream(Stream: TStream;TarFilename: String;
                    FileDateGmt: TDateTime)
```

Visibility: public

Description: `AddStream` will add the contents of `Stream` to the archive. The Stream will not be reset: only the contents of the stream from the current position will be written to the archive. The entry will be written with file name `TarFileName`. This name should contain only forward slash path separators. The entry will be written with timestamp `FileDateGmt`.

The archive entry is written with the current owner data and permissions.

See also: TTarWriter.AddFile (275), TTarWriter.AddString (276), TTarWriter.AddLink (277), TTarWriter.AddSymbolicLink (276), TTarWriter.AddDir (276), TTarWriter.AddVolumeHeader (277)

20.6.8 TTarWriter.AddString

Synopsis: Add string as file data

Declaration:
```
procedure AddString(Contents: String;TarFilename: String;
                    FileDateGmt: TDateTime)
```

Visibility: public

Description: `AddString` adds the string `Contents` as the data of an entry with file name `TarFileName`. This name should contain only forward slash path separators. The entry will be written with timestamp `FileDateGmt`.

The archive entry is written with the current owner data and permissions.

See also: TTarWriter.AddFile (275), TTarWriter.AddStream (275), TTarWriter.AddLink (277), TTarWriter.AddSymbolicLink (276), TTarWriter.AddDir (276), TTarWriter.AddVolumeHeader (277)

20.6.9 TTarWriter.AddDir

Synopsis: Add directory to archive

Declaration:
```
procedure AddDir(Dirname: String;DateGmt: TDateTime;MaxDirSize: Int64)
```

Visibility: public

Description: `AddDir` adds a directory entry to the archive. The entry is written with name `DirName`, maximum directory size `MaxDirSize` (0 means unlimited) and timestamp `DateGmt`.

Note that this call only adds an entry for a directory to the archive: if `DirName` is an existing directory, it does not write all files in the directory to the archive.

The directory entry is written with the current owner data and permissions.

See also: TTarWriter.AddFile (275), TTarWriter.AddStream (275), TTarWriter.AddLink (277), TTarWriter.AddSymbolicLink (276), TTarWriter.AddString (276), TTarWriter.AddVolumeHeader (277)

20.6.10 TTarWriter.AddSymbolicLink

Synopsis: Add a symbolic link to the archive

Declaration:
```
procedure AddSymbolicLink(Filename: String;Linkname: String;
                          DateGmt: TDateTime)
```

Visibility: public

Description: `AddSymbolicLink` adds a symbolic link entry to the archive, with name `FileName`, pointing to `LinkName`. The entry is written with timestamp `DateGmt`.

The link entry is written with the current owner data and permissions.

Errors:

See also: TTarWriter.AddFile (275), TTarWriter.AddStream (275), TTarWriter.AddLink (277), TTarWriter.AddDir (276), TTarWriter.AddString (276), TTarWriter.AddVolumeHeader (277)

20.6.11 TTarWriter.AddLink

Synopsis: Add hard link to archive

Declaration: `procedure AddLink(Filename: String;Linkname: String;DateGmt: TDateTime)`

Visibility: public

Description: `AddLink` adds a hard link entry to the archive. The entry has name `FileName`, timestamp `DateGmt` and points to `LinkName`.

The link entry is written with the current owner data and permissions.

Errors:

See also: TTarWriter.AddFile (275), TTarWriter.AddStream (275), TTarWriter.AddSymbolicLink (276), TTarWriter.AddDir (276), TTarWriter.AddString (276), TTarWriter.AddVolumeHeader (277)

20.6.12 TTarWriter.AddVolumeHeader

Synopsis: Add volume header entry

Declaration: `procedure AddVolumeHeader(VolumeId: String;DateGmt: TDateTime)`

Visibility: public

Description: `AddVolumeHeader` adds a volume header entry to the archive. The entry is written with name `VolumeID` and timestamp `DateGmt`.

The volume header entry is written with the current owner data and permissions.

Errors:

See also: TTarWriter.AddFile (275), TTarWriter.AddStream (275), TTarWriter.AddSymbolicLink (276), TTarWriter.AddDir (276), TTarWriter.AddString (276), TTarWriter.AddLink (277)

20.6.13 TTarWriter.Finalize

Synopsis: Finalize the archive

Declaration: `procedure Finalize`

Visibility: public

Description: `Finalize` writes the end-of-archive marker to the archive. No more entries can be added after `Finalize` was called.

If the `TTarWriter` instance is destroyed, it will automatically call `finalize` if `finalize` was not yet called.

See also: TTarWriter.Destroy (275)

20.6.14 TTarWriter.Permissions

Synopsis: Archive entry permissions

Declaration: `Property Permissions : TTarPermissions`

Visibility: public

Access: Read,Write

Description: `Permissions` is used for the permissions field of the archive entries.

See also: TTarDirRec (270)

20.6.15 TTarWriter.UID

Synopsis: Archive entry user ID

Declaration: `Property UID : Integer`

Visibility: public

Access: Read,Write

Description: `UID` is used for the `UID` field of the archive entries.

See also: TTarDirRec (270)

20.6.16 TTarWriter.GID

Synopsis: Archive entry group ID

Declaration: `Property GID : Integer`

Visibility: public

Access: Read,Write

Description: `GID` is used for the `GID` field of the archive entries.

See also: TTarDirRec (270)

20.6.17 TTarWriter.UserName

Synopsis: Archive entry user name

Declaration: `Property UserName : String`

Visibility: public

Access: Read,Write

Description: `UserName` is used for the `UserName` field of the archive entries.

See also: TTarDirRec (270)

20.6.18 TTarWriter.GroupName

Synopsis: Archive entry group name

Declaration: `Property GroupName : String`

Visibility: public

Access: Read,Write

Description: `GroupName` is used for the `GroupName` field of the archive entries.

See also: TTarDirRec (270)

20.6.19 TTarWriter.Mode

Synopsis: Archive entry mode

Declaration: `Property Mode : TTarModes`

Visibility: public

Access: Read,Write

Description: `Mode` is used for the `Mode` field of the archive entries.

See also: TTarDirRec (270)

20.6.20 TTarWriter.Magic

Synopsis: Archive entry Magic constant

Declaration: `Property Magic : String`

Visibility: public

Access: Read,Write

Description: `Magic` is used for the `Magic` field of the archive entries.

See also: TTarDirRec (270)

Chapter 21

Reference for unit 'Pipes'

21.1 Used units

Table 21.1: Used units by unit 'Pipes'

Name	Page
Classes	??
sysutils	??

21.2 Overview

The Pipes unit implements streams that are wrappers around the OS's pipe functionality. It creates a pair of streams, and what is written to one stream can be read from another.

21.3 Constants, types and variables

21.3.1 Constants

```
ENoSeekMsg = 'Cannot seek on pipes'
```

Constant used in EPipeSeek (281) exception.

```
EPipeMsg = 'Failed to create pipe.'
```

Constant used in EPipeCreation (281) exception.

21.4 Procedures and functions

21.4.1 CreatePipeHandles

Synopsis: Function to create a set of pipe handles

Declaration: `function CreatePipeHandles(var Inhandle: THandle;var OutHandle: THandle) : Boolean`

Visibility: default

Description: `CreatePipeHandles` provides an OS-independent way to create a set of pipe filehandles. These handles are inheritable to child processes. The reading end of the pipe is returned in `InHandle`, the writing end in `OutHandle`.

Errors: On error, `False` is returned.

See also: CreatePipeStreams (281)

21.4.2 CreatePipeStreams

Synopsis: Create a pair of pipe stream.

Declaration: `procedure CreatePipeStreams(var InPipe: TInputPipeStream; var OutPipe: TOutputPipeStream)`

Visibility: default

Description: `CreatePipeStreams` creates a set of pipe file descriptors with CreatePipeHandles (280), and if that call is succesfull, a pair of streams is created: `InPipe` and `OutPipe`.

Errors: If no pipe handles could be created, an EPipeCreation (281) exception is raised.

See also: CreatePipeHandles (280), TInputPipeStream (281), TOutputPipeStream (283)

21.5 EPipeCreation

21.5.1 Description

Exception raised when an error occurred during the creation of a pipe pair.

21.6 EPipeError

21.6.1 Description

Exception raised when an invalid operation is performed on a pipe stream.

21.7 EPipeSeek

21.7.1 Description

Exception raised when an invalid seek operation is attempted on a pipe.

21.8 TInputPipeStream

21.8.1 Description

`TInputPipeStream` is created by the CreatePipeStreams (281) call to represent the reading end of a pipe. It is a TStream (**??**) descendent which does not allow writing, and which mimics the seek operation.

21.8.2 Method overview

Page	Property	Description
283	Read	Read data from the stream to a buffer.
282	Seek	Set the current position of the stream
282	Write	Write data to the stream.

21.8.3 Property overview

Page	Property	Access	Description
283	NumBytesAvailable	r	Number of bytes available for reading.

21.8.4 TInputPipeStream.Write

Synopsis: Write data to the stream.

Declaration: `function Write(const Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Write` overrides the parent implementation of `Write`. On a `TInputPipeStream` will always raise an exception, as the pipe is read-only.

Errors: An ENoWritePipe (280) exception is raised when this function is called.

See also: TInputPipeStream.Read (283), TInputPipeStream.Seek (282)

21.8.5 TInputPipeStream.Seek

Synopsis: Set the current position of the stream

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` overrides the standard `Seek` implementation. Normally, pipe streams stderr are not seekable. The `TInputPipeStream` stream tries to provide seek capabilities for the following limited number of cases:

Origin=soFromBeginning If `Offset` is larger than the current position, then the remaining bytes are skipped by reading them from the stream and discarding them.

Origin=soFromCurrent If `Offset` is zero, the current position is returned. If it is positive, then `Offset` bytes are skipped by reading them from the stream and discarding them, if the stream is of type `iosInput`.

All other cases will result in a `EPipeSeek` exception.

Errors: An EPipeSeek (281) exception is raised if the stream does not allow the requested seek operation.

See also: EPipeSeek (281), #rtl.classes.tstream.seek (**??**)

21.8.6 TInputPipeStream.Read

Synopsis: Read data from the stream to a buffer.

Declaration: `function Read(var Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` calls the inherited read and adjusts the internal position pointer of the stream.

Errors: None.

See also: TInputPipeStream.Write (282), TInputPipeStream.Seek (282)

21.8.7 TInputPipeStream.NumBytesAvailable

Synopsis: Number of bytes available for reading.

Declaration: `Property NumBytesAvailable : DWord`

Visibility: public

Access: Read

Description: `NumBytesAvailable` is the number of bytes available for reading. This is the number of bytes in the OS buffer for the pipe. It is not a number of bytes in an internal buffer.

If this number is nonzero, then reading `NumBytesAvailable` bytes from the stream will not block the process. Reading more than `NumBytesAvailable` bytes will block the process, while it waits for the requested number of bytes to become available.

See also: TInputPipeStream.Read (283)

21.9 TOutputPipeStream

21.9.1 Description

`TOutputPipeStream` is created by the CreatePipeStreams (281) call to represent the writing end of a pipe. It is a TStream (**??**) descendent which does not allow reading.

21.9.2 Method overview

Page	Property	Description
284	Read	Read data from the stream.
283	Seek	Sets the position in the stream

21.9.3 TOutputPipeStream.Seek

Synopsis: Sets the position in the stream

Declaration: `function Seek(Offset: LongInt;Origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` is overridden in `TOutputPipeStream`. Calling this method will always raise an exception: an output pipe is not seekable.

Errors: An EPipeSeek (281) exception is raised if this method is called.

21.9.4 TOutputPipeStream.Read

Synopsis: Read data from the stream.

Declaration: `function Read(var Buffer;Count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` overrides the parent `Read` implementation. It always raises an exception, because a output pipe is write-only.

Errors: An ENoReadPipe (280) exception is raised when this function is called.

See also: TOutputPipeStream.Seek (283)

Chapter 22

Reference for unit 'pooledmm'

22.1 Used units

Table 22.1: Used units by unit 'pooledmm'

Name	Page
Classes	??

22.2 Overview

pooledmm is a memory manager class which uses pools of blocks. Since it is a higher-level implementation of a memory manager which works on top of the FPC memory manager, It also offers more debugging and analysis tools. It is used mainly in the LCL and Lazarus IDE.

22.3 Constants, types and variables

22.3.1 Types

```
PPooledMemManagerItem = ^TPooledMemManagerItem
```

`PPooledMemManagerItem` is a pointer type, pointing to a TPooledMemManagerItem (286) item, used in a linked list.

```
TEnumItemsMethod = procedure(Item: Pointer) of object
```

`TEnumItemsMethod` is a prototype for the callback used in the TNonFreePooledMemManager.EnumerateItems (287) call. The parameter `Item` will be set to each of the pointers in the item list of TNonFreePooledMemManager (286).

```
TPooledMemManagerItem = record
  Next : PPooledMemManagerItem;
end
```

TPooledMemManagerItem is used internally by the TPooledMemManager (288) class to maintain the free list block. It simply points to the next free block.

22.4 TNonFreePooledMemManager

22.4.1 Description

TNonFreePooledMemManager keeps a list of fixed-size memory blocks in memory. Each block has the same size, making it suitable for storing a lot of records of the same type. It does not free the items stored in it, except when the list is cleared as a whole.

It allocates memory for the blocks in a exponential way, i.e. each time a new block of memory must be allocated, it's size is the double of the last block. The first block will contain 8 items.

22.4.2 Method overview

Page	Property	Description
286	Clear	Clears the memory
286	Create	Creates a new instance of TNonFreePooledMemManager
287	Destroy	Removes the TNonFreePooledMemManager instance from memory
287	EnumerateItems	Enumerate all items in the list
287	NewItem	Return a pointer to a new memory block

22.4.3 Property overview

Page	Property	Access	Description
287	ItemSize	r	Size of an item in the list

22.4.4 TNonFreePooledMemManager.Clear

Synopsis: Clears the memory

Declaration: `procedure Clear`

Visibility: public

Description: Clear clears all blocks from memory, freeing the allocated memory blocks. None of the pointers returned by NewItem (287) is valid after a call to Clear

See also: TNonFreePooledMemManager.NewItem (287)

22.4.5 TNonFreePooledMemManager.Create

Synopsis: Creates a new instance of TNonFreePooledMemManager

Declaration: `constructor Create(TheItemSize: Integer)`

Visibility: public

Description: Create creates a new instance of TNonFreePooledMemManager and sets the item size to TheItemSize.

Errors: If not enough memory is available, an exception may be raised.

See also: TNonFreePooledMemManager.ItemSize (287)

22.4.6 TNonFreePooledMemManager.Destroy

Synopsis: Removes the `TNonFreePooledMemManager` instance from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` clears the list, clears the internal structures, and then calls the inherited `Destroy`.

`Destroy` should never be called directly. Instead `Free` should be used, or `FreeAndNil`

See also: TNonFreePooledMemManager.Create (286), TNonFreePooledMemManager.Clear (286)

22.4.7 TNonFreePooledMemManager.NewItem

Synopsis: Return a pointer to a new memory block

Declaration: `function NewItem : Pointer`

Visibility: public

Description: `NewItem` returns a pointer to an unused memory block of size ItemSize (287). It will allocate new memory on the heap if necessary.

Note that there is no way to mark the memory block as free, except by clearing the whole list.

Errors: If no more memory is available, an exception may be raised.

See also: TNonFreePooledMemManager.Clear (286)

22.4.8 TNonFreePooledMemManager.EnumerateItems

Synopsis: Enumerate all items in the list

Declaration: `procedure EnumerateItems(const Method: TEnumItemsMethod)`

Visibility: public

Description: `EnumerateItems` will enumerate over all items in the list, passing the items to `Method`. This can be used to execute certain operations on all items in the list. (for example, simply list them)

22.4.9 TNonFreePooledMemManager.ItemSize

Synopsis: Size of an item in the list

Declaration: `Property ItemSize : Integer`

Visibility: public

Access: Read

Description: `ItemSize` is the size of a single block in the list. It's a fixed size determined when the list is created.

See also: TNonFreePooledMemManager.Create (286)

22.5 TPooledMemManager

22.5.1 Description

`TPooledMemManager` is a class which maintains a linked list of blocks, represented by the TPooledMemManagerItem (286) record. It should not be used directly, but should be descended from and the descendent should implement the actual memory manager.

22.5.2 Method overview

Page	Property	Description
288	Clear	Clears the list
288	Create	Creates a new instance of the `TPooledMemManager` class
288	Destroy	Removes an instance of `TPooledMemManager` class from memory

22.5.3 Property overview

Page	Property	Access	Description
290	AllocatedCount	r	Total number of allocated items in the list
289	Count	r	Number of items in the list
290	FreeCount	r	Number of free items in the list
290	FreedCount	r	Total number of freed items in the list.
289	MaximumFreeCountRatio	rw	Maximum ratio of free items over total items
289	MinimumFreeCount	rw	Minimum count of free items in the list

22.5.4 TPooledMemManager.Clear

Synopsis: Clears the list

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` clears the list, it disposes all items in the list.

See also: TPooledMemManager.FreedCount (290)

22.5.5 TPooledMemManager.Create

Synopsis: Creates a new instance of the `TPooledMemManager` class

Declaration: `constructor Create`

Visibility: public

Description: `Create` initializes all necessary properties and then calls the inherited create.

See also: TPooledMemManager.Destroy (288)

22.5.6 TPooledMemManager.Destroy

Synopsis: Removes an instance of `TPooledMemManager` class from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` calls Clear (288) and then calls the inherited destroy.

`Destroy` should never be called directly. Instead `Free` should be used, or `FreeAndNil`

See also: TPooledMemManager.Create (288)

22.5.7 TPooledMemManager.MinimumFreeCount

Synopsis: Minimum count of free items in the list

Declaration: `Property MinimumFreeCount : Integer`

Visibility: public

Access: Read,Write

Description: `MinimumFreeCount` is the minimum number of free items in the linked list. When disposing an item in the list, the number of items is checked, and only if the required number of free items is present, the item is actually freed.

The default value is 100000

See also: TPooledMemManager.MaximumFreeCountRatio (289)

22.5.8 TPooledMemManager.MaximumFreeCountRatio

Synopsis: Maximum ratio of free items over total items

Declaration: `Property MaximumFreeCountRatio : Integer`

Visibility: public

Access: Read,Write

Description: `MaximumFreeCountRatio` is the maximum ratio (divided by 8) of free elements over the total amount of elements: When disposing an item in the list, if the number of free items is higher than this ratio, the item is freed.

The default value is 8.

See also: TPooledMemManager.MinimumFreeCount (289)

22.5.9 TPooledMemManager.Count

Synopsis: Number of items in the list

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the total number of items allocated from the list.

See also: TPooledMemManager.FreeCount (290), TPooledMemManager.AllocatedCount (290), TPooledMemManager.FreedCount (290)

22.5.10 TPooledMemManager.FreeCount

Synopsis: Number of free items in the list

Declaration: `Property FreeCount : Integer`

Visibility: public

Access: Read

Description: `FreeCount` is the current total number of free items in the list.

See also: TPooledMemManager.Count (289), TPooledMemManager.AllocatedCount (290), TPooledMemManager.FreedCount (290)

22.5.11 TPooledMemManager.AllocatedCount

Synopsis: Total number of allocated items in the list

Declaration: `Property AllocatedCount : Int64`

Visibility: public

Access: Read

Description: `AllocatedCount` is the total number of newly allocated items on the list.

See also: TPooledMemManager.Count (289), TPooledMemManager.FreeCount (290), TPooledMemManager.FreedCount (290)

22.5.12 TPooledMemManager.FreedCount

Synopsis: Total number of freed items in the list.

Declaration: `Property FreedCount : Int64`

Visibility: public

Access: Read

Description: `FreedCount` is the total number of elements actually freed in the list.

See also: TPooledMemManager.Count (289), TPooledMemManager.FreeCount (290), TPooledMemManager.AllocatedCount (290)

Chapter 23

Reference for unit 'process'

23.1 Used units

Table 23.1: Used units by unit 'process'

Name	Page
Classes	??
Pipes	280
sysutils	??

23.2 Overview

The Process unit contains the code for the TProcess (293) component, a cross-platform component to start and control other programs, offering also access to standard input and output for these programs.

`TProcess` does not handle wildcard expansion, does not support complex pipelines as in Unix. If this behaviour is desired, the shell can be executed with the pipeline as the command it should execute.

23.3 Constants, types and variables

23.3.1 Types

```
TProcessOption = (poRunSuspended,poWaitOnExit,poUsePipes,
                  poStderrToOutPut,poNoConsole,poNewConsole,
                  poDefaultErrorMode,poNewProcessGroup,poDebugProcess,
                  poDebugOnlyThisProcess)
```

When a new process is started using TProcess.Execute (295), these options control the way the process is started. Note that not all options are supported on all platforms.

```
TProcessOptions= Set of (poDebugOnlyThisProcess,poDebugProcess,
                         poDefaultErrorMode,poNewConsole,
```

Table 23.2: Enumeration values for type TProcessOption

Value	Explanation
poDebugOnlyThisProcess	Do not follow processes started by this process (Win32 only)
poDebugProcess	Allow debugging of the process (Win32 only)
poDefaultErrorMode	Use default error handling.
poNewConsole	Start a new console window for the process (Win32 only)
poNewProcessGroup	Start the process in a new process group (Win32 only)
poNoConsole	Do not allow access to the console window for the process (Win32 only)
poRunSuspended	Start the process in suspended state.
poStderrToOutPut	Redirect standard error to the standard output stream.
poUsePipes	Use pipes to redirect standard input and output.
poWaitOnExit	Wait for the process to terminate before returning.

```
                                poNewProcessGroup,poNoConsole,poRunSuspended,
                                poStderrToOutPut,poUsePipes,poWaitOnExit)
```

Set of TProcessOption (291).

```
TProcessPriority = (ppHigh,ppIdle,ppNormal,ppRealTime)
```

Table 23.3: Enumeration values for type TProcessPriority

Value	Explanation
ppHigh	The process runs at higher than normal priority.
ppIdle	The process only runs when the system is idle (i.e. has nothing else to do)
ppNormal	The process runs at normal priority.
ppRealTime	The process runs at real-time priority.

This enumerated type determines the priority of the newly started process. It translates to default platform specific constants. If finer control is needed, then platform-dependent mechanism need to be used to set the priority.

```
TShowWindowOptions = (swoNone,swoHIDE,swoMaximize,swoMinimize,
                      swoRestore,swoShow,swoShowDefault,
                      swoShowMaximized,swoShowMinimized,
                      swoshowMinNOActive,swoShowNA,swoShowNoActivate,
                      swoShowNormal)
```

This type describes what the new process' main window should look like. Most of these have only effect on Windows. They are ignored on other systems.

```
TStartupOption = (suoUseShowWindow,suoUseSize,suoUsePosition,
                  suoUseCountChars,suoUseFillAttribute)
```

These options are mainly for Win32, and determine what should be done with the application once it's started.

Table 23.4: Enumeration values for type TShowWindowOptions

Value	Explanation
swoHIDE	The main window is hidden.
swoMaximize	The main window is maximized.
swoMinimize	The main window is minimized.
swoNone	Allow system to position the window.
swoRestore	Restore the previous position.
swoShow	Show the main window.
swoShowDefault	When showing Show the main window on
swoShowMaximized	The main window is shown maximized
swoShowMinimized	The main window is shown minimized
swoshowMinNOActive	The main window is shown minimized but not activated
swoShowNA	The main window is shown but not activated
swoShowNoActivate	The main window is shown but not activated
swoShowNormal	The main window is shown normally

Table 23.5: Enumeration values for type TStartupOption

Value	Explanation
suoUseCountChars	Use the console character width as specified in TProcess (293).
suoUseFillAttribute	Use the console fill attribute as specified in TProcess (293).
suoUsePosition	Use the window sizes as specified in TProcess (293).
suoUseShowWindow	Use the Show Window options specified in TShowWindowOption (292)
suoUseSize	Use the window sizes as specified in TProcess (293)

```
TStartupOptions= Set of (suoUseCountChars,suoUseFillAttribute,
                         suoUsePosition,suoUseShowWindow,suoUseSize)
```

Set of TStartUpOption (292).

23.4 EProcess

23.4.1 Description

Exception raised when an error occurs in a TProcess routine.

23.5 TProcess

23.5.1 Description

`TProcess` is a component that can be used to start and control other processes (programs/binaries). It contains a lot of options that control how the process is started. Many of these are Win32 specific, and have no effect on other platforms, so they should be used with care.

The simplest way to use this component is to create an instance, set the CommandLine (301) property to the full pathname of the program that should be executed, and call Execute (295). To determine whether the process is still running (i.e. has not stopped executing), the Running (305) property can be checked.

More advanced techniques can be used with the Options (303) settings.

23.5.2 Method overview

Page	Property	Description
296	CloseInput	Close the input stream of the process
296	CloseOutput	Close the output stream of the process
296	CloseStderr	Close the error stream of the process
295	Create	Create a new instance of the `TProcess` class.
295	Destroy	Destroy this instance of `TProcess`
295	Execute	Execute the program with the given options
296	Resume	Resume execution of a suspended process
297	Suspend	Suspend a running process
297	Terminate	Terminate a running process
297	WaitOnExit	Wait for the program to stop executing.

23.5.3 Property overview

Page	Property	Access	Description
301	Active	rw	Start or stop the process.
301	ApplicationName	rw	Name of the application to start
301	CommandLine	rw	Command-line to execute
302	ConsoleTitle	rw	Title of the console window
302	CurrentDirectory	rw	Working directory of the process.
302	Desktop	rw	Desktop on which to start the process.
303	Environment	rw	Environment variables for the new process
300	ExitStatus	r	Exit status of the process.
308	FillAttribute	rw	Color attributes of the characters in the console window (Windows only)
298	Handle	r	Handle of the process
301	InheritHandles	rw	Should the created process inherit the open handles of the current process.
299	Input	r	Stream connected to standard input of the process.
303	Options	rw	Options to be used when starting the process.
300	Output	r	Stream connected to standard output of the process.
304	Priority	rw	Priority at which the process is running.
298	ProcessHandle	r	Alias for Handle (298)
299	ProcessID	r	ID of the process.
305	Running	r	Determines wheter the process is still running.
305	ShowWindow	rw	Determines how the process main window is shown (Windows only)
304	StartupOptions	rw	Additional (Windows) startup options
300	Stderr	r	Stream connected to standard diagnostic output of the process.
298	ThreadHandle	r	Main process thread handle
299	ThreadID	r	ID of the main process thread
306	WindowColumns	rw	Number of columns in console window (windows only)
306	WindowHeight	rw	Height of the process main window
306	WindowLeft	rw	X-coordinate of the initial window (Windows only)
298	WindowRect	rw	Positions for the main program window.
307	WindowRows	rw	Number of rows in console window (Windows only)
307	WindowTop	rw	Y-coordinate of the initial window (Windows only)
307	WindowWidth	rw	Height of the process main window (Windows only)

23.5.4 TProcess.Create

Synopsis: Create a new instance of the `TProcess` class.

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` creates a new instance of the `TProcess` class. After calling the inherited constructor, it simply sets some default values.

23.5.5 TProcess.Destroy

Synopsis: Destroy this instance of `TProcess`

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` cleans up this instance of `TProcess`. Prior to calling the inherited destructor, it cleans up any streams that may have been created. If a process was started and is still executed, it is *not* stopped, but the standard input/output/stderr streams are no longer available, because they have been destroyed.

Errors: None.

See also: TProcess.Create (295)

23.5.6 TProcess.Execute

Synopsis: Execute the program with the given options

Declaration: `procedure Execute; Virtual`

Visibility: public

Description: `Execute` actually executes the program as specified in CommandLine (301), applying as much as of the specified options as supported on the current platform.

If the `poWaitOnExit` option is specified in Options (303), then the call will only return when the program has finished executing (or if an error occured). If this option is not given, the call returns immediatly, but the WaitOnExit (297) call can be used to wait for it to close, or the Running (305) call can be used to check whether it is still running.

The TProcess.Terminate (297) call can be used to terminate the program if it is still running, or the Suspend (297) call can be used to temporarily stop the program's execution.

The ExitStatus (300) function can be used to check the program's exit status, after it has stopped executing.

Errors: On error a EProcess (293) exception is raised.

See also: TProcess.Running (305), TProcess.WaitOnExit (297), TProcess.Terminate (297), TProcess.Suspend (297), TProcess.Resume (296), TProcess.ExitStatus (300)

23.5.7 TProcess.CloseInput

Synopsis: Close the input stream of the process

Declaration: `procedure CloseInput; Virtual`

Visibility: public

Description: `CloseInput` closes the input file descriptor of the process, that is, it closes the handle of the pipe to standard input of the process.

See also: TProcess.Input (299), TProcess.StdErr (300), TProcess.Output (300), TProcess.CloseOutput (296), TProcess.CloseStdErr (296)

23.5.8 TProcess.CloseOutput

Synopsis: Close the output stream of the process

Declaration: `procedure CloseOutput; Virtual`

Visibility: public

Description: `CloseOutput` closes the output file descriptor of the process, that is, it closes the handle of the pipe to standard output of the process.

See also: TProcess.Output (300), TProcess.Input (299), TProcess.StdErr (300), TProcess.CloseInput (296), TProcess.CloseStdErr (296)

23.5.9 TProcess.CloseStderr

Synopsis: Close the error stream of the process

Declaration: `procedure CloseStderr; Virtual`

Visibility: public

Description: `CloseStdErr` closes the standard error file descriptor of the process, that is, it closes the handle of the pipe to standard error output of the process.

See also: TProcess.Output (300), TProcess.Input (299), TProcess.StdErr (300), TProcess.CloseInput (296), TProcess.CloseStdErr (296)

23.5.10 TProcess.Resume

Synopsis: Resume execution of a suspended process

Declaration: `function Resume : Integer; Virtual`

Visibility: public

Description: `Resume` should be used to let a suspended process resume it's execution. It should be called in particular when the `poRunSuspended` flag is set in Options (303).

Errors: None.

See also: TProcess.Suspend (297), TProcess.Options (303), TProcess.Execute (295), TProcess.Terminate (297)

23.5.11 TProcess.Suspend

Synopsis: Suspend a running process

Declaration: `function Suspend : Integer;  Virtual`

Visibility: public

Description: `Suspend` suspends a running process. If the call is successful, the process is suspended: it stops running, but can be made to execute again using the Resume (296) call.

`Suspend` is fundamentally different from TProcess.Terminate (297) which actually stops the process.

Errors: On error, a nonzero result is returned.

See also: TProcess.Options (303), TProcess.Resume (296), TProcess.Terminate (297), TProcess.Execute (295)

23.5.12 TProcess.Terminate

Synopsis: Terminate a running process

Declaration: `function Terminate(AExitCode: Integer) : Boolean;  Virtual`

Visibility: public

Description: `Terminate` stops the execution of the running program. It effectively stops the program.

On Windows, the program will report an exit code of `AExitCode`, on other systems, this value is ignored.

Errors: On error, a nonzero value is returned.

See also: TProcess.ExitStatus (300), TProcess.Suspend (297), TProcess.Execute (295), TProcess.WaitOnExit (297)

23.5.13 TProcess.WaitOnExit

Synopsis: Wait for the program to stop executing.

Declaration: `function WaitOnExit : Boolean`

Visibility: public

Description: `WaitOnExit` waits for the running program to exit. It returns `True` if the wait was succesful, or `False` if there was some error waiting for the program to exit.

Note that the return value of this function has changed. The old return value was a DWord with a platform dependent error code. To make things consistent and cross-platform, a boolean return type was used.

Errors: On error, `False` is returned. No extended error information is available, as it is highly system dependent.

See also: TProcess.ExitStatus (300), TProcess.Terminate (297), TProcess.Running (305)

23.5.14 TProcess.WindowRect

Synopsis: Positions for the main program window.

Declaration: `Property WindowRect : Trect`

Visibility: public

Access: Read,Write

Description: `WindowRect` can be used to specify the position of

23.5.15 TProcess.Handle

Synopsis: Handle of the process

Declaration: `Property Handle : THandle`

Visibility: public

Access: Read

Description: `Handle` identifies the process. In Unix systems, this is the process ID. On windows, this is the process handle. It can be used to signal the process.

The handle is only valid after TProcess.Execute (295) has been called. It is not reset after the process stopped.

See also: TProcess.ThreadHandle (298), TProcess.ProcessID (299), TProcess.ThreadID (299)

23.5.16 TProcess.ProcessHandle

Synopsis: Alias for Handle (298)

Declaration: `Property ProcessHandle : THandle`

Visibility: public

Access: Read

Description: `ProcessHandle` equals Handle (298) and is provided for completeness only.

See also: TProcess.Handle (298), TProcess.ThreadHandle (298), TProcess.ProcessID (299), TProcess.ThreadID (299)

23.5.17 TProcess.ThreadHandle

Synopsis: Main process thread handle

Declaration: `Property ThreadHandle : THandle`

Visibility: public

Access: Read

Description: `ThreadHandle` is the main process thread handle. On Unix, this is the same as the process ID, on Windows, this may be a different handle than the process handle.

The handle is only valid after TProcess.Execute (295) has been called. It is not reset after the process stopped.

See also: TProcess.Handle (298), TProcess.ProcessID (299), TProcess.ThreadID (299)

23.5.18 TProcess.ProcessID

Synopsis: ID of the process.

Declaration: `Property ProcessID : Integer`

Visibility: public

Access: Read

Description: `ProcessID` is the ID of the process. It is the same as the handle of the process on Unix systems, but on Windows it is different from the process Handle.

The ID is only valid after TProcess.Execute (295) has been called. It is not reset after the process stopped.

See also: TProcess.Handle (298), TProcess.ThreadHandle (298), TProcess.ThreadID (299)

23.5.19 TProcess.ThreadID

Synopsis: ID of the main process thread

Declaration: `Property ThreadID : Integer`

Visibility: public

Access: Read

Description: `ProcessID` is the ID of the main process thread. It is the same as the handle of the main proces thread (or the process itself) on Unix systems, but on Windows it is different from the thread Handle.

The ID is only valid after TProcess.Execute (295) has been called. It is not reset after the process stopped.

See also: TProcess.ProcessID (299), TProcess.Handle (298), TProcess.ThreadHandle (298)

23.5.20 TProcess.Input

Synopsis: Stream connected to standard input of the process.

Declaration: `Property Input : TOutputPipeStream`

Visibility: public

Access: Read

Description: `Input` is a stream which is connected to the process' standard input file handle. Anything written to this stream can be read by the process.

The `Input` stream is only instantiated when the `poUsePipes` flag is used in Options (303).

Note that writing to the stream may cause the calling process to be suspended when the created process is not reading from it's input, or to cause errors when the process has terminated.

See also: TProcess.OutPut (300), TProcess.StdErr (300), TProcess.Options (303), TProcessOption (291)

23.5.21 TProcess.Output

Synopsis: Stream connected to standard output of the process.

Declaration: `Property Output : TInputPipeStream`

Visibility: public

Access: Read

Description: `Output` is a stream which is connected to the process' standard output file handle. Anything written to standard output by the created process can be read from this stream.

The `Output` stream is only instantiated when the `poUsePipes` flag is used in Options (303).

The `Output` stream also contains any data written to standard diagnostic output (`stderr`) when the `poStdErrToOutPut` flag is used in Options (303).

Note that reading from the stream may cause the calling process to be suspended when the created process is not writing anything to standard output, or to cause errors when the process has terminated.

See also: TProcess.InPut (299), TProcess.StdErr (300), TProcess.Options (303), TProcessOption (291)

23.5.22 TProcess.Stderr

Synopsis: Stream connected to standard diagnostic output of the process.

Declaration: `Property Stderr : TInputPipeStream`

Visibility: public

Access: Read

Description: `StdErr` is a stream which is connected to the process' standard diagnostic output file handle (`StdErr`). Anything written to standard diagnostic output by the created process can be read from this stream.

The `StdErr` stream is only instantiated when the `poUsePipes` flag is used in Options (303).

The `Output` stream equals the Output (300) when the `poStdErrToOutPut` flag is used in Options (303).

Note that reading from the stream may cause the calling process to be suspended when the created process is not writing anything to standard output, or to cause errors when the process has terminated.

See also: TProcess.InPut (299), TProcess.Output (300), TProcess.Options (303), TProcessOption (291)

23.5.23 TProcess.ExitStatus

Synopsis: Exit status of the process.

Declaration: `Property ExitStatus : Integer`

Visibility: public

Access: Read

Description: `ExitStatus` contains the exit status as reported by the process when it stopped executing. The value of this property is only meaningful when the process is no longer running. If it is not running then the value is zero.

See also: TProcess.Running (305), TProcess.Terminate (297)

23.5.24 TProcess.InheritHandles

Synopsis: Should the created process inherit the open handles of the current process.

Declaration: `Property InheritHandles : Boolean`

Visibility: public

Access: Read,Write

Description: `InheritHandles` determines whether the created process inherits the open handles of the current process (value `True`) or not (`False`).

On Unix, setting this variable has no effect.

See also: TProcess.InPut (299), TProcess.Output (300), TProcess.StdErr (300)

23.5.25 TProcess.Active

Synopsis: Start or stop the process.

Declaration: `Property Active : Boolean`

Visibility: published

Access: Read,Write

Description: `Active` starts the process if it is set to `True`, or terminates the process if set to `False`. It's mostly intended for use in an IDE.

See also: TProcess.Execute (295), TProcess.Terminate (297)

23.5.26 TProcess.ApplicationName

Synopsis: Name of the application to start

Declaration: `Property ApplicationName : String`

Visibility: published

Access: Read,Write

Description: `ApplicationName` is an alias for TProcess.CommandLine (301). It's mostly foruse in the Windows `CreateProcess` call. If `CommandLine` is not set, then `ApplicationName` will be used instead.

Note that either `CommandLine` or `ApplicationName` must be set prior to calling `Execute`.

See also: TProcess.CommandLine (301)

23.5.27 TProcess.CommandLine

Synopsis: Command-line to execute

Declaration: `Property CommandLine : String`

Visibility: published

Access: Read,Write

Description: `CommandLine` is the command-line to be executed: this is the name of the program to be executed, followed by any options it should be passed.

If the command to be executed or any of the arguments contains whitespace (space, tab character, linefeed character) it should be enclosed in single or double quotes.

If no absolute pathname is given for the command to be executed, it is searched for in the `PATH` environment variable. On Windows, the current directory always will be searched first. On other platforms, this is not so.

Note that either `CommandLine` or `ApplicationName` must be set prior to calling `Execute`.

See also: TProcess.ApplicationName (301)

23.5.28 TProcess.ConsoleTitle

Synopsis: Title of the console window

Declaration: `Property ConsoleTitle : String`

Visibility: published

Access: Read,Write

Description: `ConsoleTitle` is used on Windows when executing a console application: it specifies the title caption of the console window. On other platforms, this property is currently ignored.

Changing this property after the process was started has no effect.

See also: TProcess.WindowColumns (306), TProcess.WindowRows (307)

23.5.29 TProcess.CurrentDirectory

Synopsis: Working directory of the process.

Declaration: `Property CurrentDirectory : String`

Visibility: published

Access: Read,Write

Description: `CurrentDirectory` specifies the working directory of the newly started process.

Changing this property after the process was started has no effect.

See also: TProcess.Environment (303)

23.5.30 TProcess.Desktop

Synopsis: Desktop on which to start the process.

Declaration: `Property Desktop : String`

Visibility: published

Access: Read,Write

Description: `DeskTop` is used on Windows to determine on which desktop the process' main window should be shown. Leaving this empty means the process is started on the same desktop as the currently running process.

Changing this property after the process was started has no effect.

On unix, this parameter is ignored.

See also: TProcess.Input (299), TProcess.Output (300), TProcess.StdErr (300)

23.5.31 TProcess.Environment

Synopsis: Environment variables for the new process

Declaration: `Property Environment : TStrings`

Visibility: published

Access: Read,Write

Description: `Environment` contains the environment for the new process; it's a list of `Name=Value` pairs, one per line.

If it is empty, the environment of the current process is passed on to the new process.

See also: TProcess.Options (303)

23.5.32 TProcess.Options

Synopsis: Options to be used when starting the process.

Declaration: `Property Options : TProcessOptions`

Visibility: published

Access: Read,Write

Description: `Options` determine how the process is started. They should be set before the Execute (295) call is made.

Table 23.6:

option	Meaning
`poRunSuspended`	Start the process in suspended state.
`poWaitOnExit`	Wait for the process to terminate before returning.
`poUsePipes`	Use pipes to redirect standard input and output.
`poStderrToOutPut`	Redirect standard error to the standard output stream.
`poNoConsole`	Do not allow access to the console window for the process (Win32 only)
`poNewConsole`	Start a new console window for the process (Win32 only)
`poDefaultErrorMode`	Use default error handling.
`poNewProcessGroup`	Start the process in a new process group (Win32 only)
`poDebugProcess`	Allow debugging of the process (Win32 only)
`poDebugOnlyThisProcess`	Do not follow processes started by this process (Win32 only)

See also: TProcessOption (291), TProcessOptions (292), TProcess.Priority (304), TProcess.StartUpOptions (304)

23.5.33 TProcess.Priority

Synopsis: Priority at which the process is running.

Declaration: `Property Priority : TProcessPriority`

Visibility: published

Access: Read,Write

Description: `Priority` determines the priority at which the process is running.

Table 23.7:

Priority	Meaning
`ppHigh`	The process runs at higher than normal priority.
`ppIdle`	The process only runs when the system is idle (i.e. has nothing else to do)
`ppNormal`	The process runs at normal priority.
`ppRealTime`	The process runs at real-time priority.

Note that not all priorities can be set by any user. Usually, only users witha dministrative rights (the root user on Unix) can set a higher process priority.

On unix, the process priority is mapped on `Nice` values as follows:

Table 23.8:

Priority	Nice value
`ppHigh`	20
`ppIdle`	20
`ppNormal`	0
`ppRealTime`	-20

See also: TProcessPriority (292)

23.5.34 TProcess.StartupOptions

Synopsis: Additional (Windows) startup options

Declaration: `Property StartupOptions : TStartupOptions`

Visibility: published

Access: Read,Write

Description: `StartUpOptions` contains additional startup options, used mostly on Windows system. They determine which other window layout properties are taken into account when starting the new process.

See also: TProcess.ShowWindow (305), TProcess.WindowHeight (306), TProcess.WindowWidth (307), TProcess.WindowLeft (306), TProcess.WindowTop (307), TProcess.WindowColumns (306), TProcess.WindowRows (307), TProcess.FillAttribute (308)

Table 23.9:

Priority	Meaning
suoUseShowWindow	Use the Show Window options specified in ShowWindow (305)
suoUseSize	Use the specified window sizes
suoUsePosition	Use the specified window sizes.
suoUseCountChars	Use the specified console character width.
suoUseFillAttribute	Use the console fill attribute specified in FillAttribute (308).

23.5.35 TProcess.Running

Synopsis: Determines wheter the process is still running.

Declaration: `Property Running : Boolean`

Visibility: published

Access: Read

Description: `Running` can be read to determine whether the process is still running.

See also: TProcess.Terminate (297), TProcess.Active (301), TProcess.ExitStatus (300)

23.5.36 TProcess.ShowWindow

Synopsis: Determines how the process main window is shown (Windows only)

Declaration: `Property ShowWindow : TShowWindowOptions`

Visibility: published

Access: Read,Write

Description: `ShowWindow` determines how the process' main window is shown. It is useful only on Windows.

Table 23.10:

Option	Meaning
swoNone	Allow system to position the window.
swoHIDE	The main window is hidden.
swoMaximize	The main window is maximized.
swoMinimize	The main window is minimized.
swoRestore	Restore the previous position.
swoShow	Show the main window.
swoShowDefault	When showing Show the main window on a default position
swoShowMaximized	The main window is shown maximized
swoShowMinimized	The main window is shown minimized
swoshowMinNOActive	The main window is shown minimized but not activated
swoShowNA	The main window is shown but not activated
swoShowNoActivate	The main window is shown but not activated
swoShowNormal	The main window is shown normally

23.5.37 TProcess.WindowColumns

Synopsis: Number of columns in console window (windows only)

Declaration: `Property WindowColumns : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowColumns` is the number of columns in the console window, used to run the command in. This property is only effective if `suoUseCountChars` is specified in StartupOptions (304)

See also: TProcess.WindowHeight (306), TProcess.WindowWidth (307), TProcess.WindowLeft (306), TProcess.WindowTop (307), TProcess.WindowRows (307), TProcess.FillAttribute (308), TProcess.StartupOptions (304)

23.5.38 TProcess.WindowHeight

Synopsis: Height of the process main window

Declaration: `Property WindowHeight : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowHeight` is the initial height (in pixels) of the process' main window. This property is only effective if `suoUseSize` is specified in StartupOptions (304)

See also: TProcess.WindowWidth (307), TProcess.WindowLeft (306), TProcess.WindowTop (307), TProcess.WindowColumns (306), TProcess.WindowRows (307), TProcess.FillAttribute (308), TProcess.StartupOptions (304)

23.5.39 TProcess.WindowLeft

Synopsis: X-coordinate of the initial window (Windows only)

Declaration: `Property WindowLeft : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowLeft` is the initial X coordinate (in pixels) of the process' main window, relative to the left border of the desktop. This property is only effective if `suoUsePosition` is specified in StartupOptions (304)

See also: TProcess.WindowHeight (306), TProcess.WindowWidth (307), TProcess.WindowTop (307), TProcess.WindowColumns (306), TProcess.WindowRows (307), TProcess.FillAttribute (308), TProcess.StartupOptions (304)

23.5.40 TProcess.WindowRows

Synopsis: Number of rows in console window (Windows only)

Declaration: `Property WindowRows : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowRows` is the number of rows in the console window, used to run the command in. This property is only effective if `suoUseCountChars` is specified in StartupOptions (304)

See also: TProcess.WindowHeight (306), TProcess.WindowWidth (307), TProcess.WindowLeft (306), TProcess.WindowTop (307), TProcess.WindowColumns (306), TProcess.FillAttribute (308), TProcess.StartupOptions (304)

23.5.41 TProcess.WindowTop

Synopsis: Y-coordinate of the initial window (Windows only)

Declaration: `Property WindowTop : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowTop` is the initial Y coordinate (in pixels) of the process' main window, relative to the top border of the desktop. This property is only effective if `suoUsePosition` is specified in StartupOptions (304)

See also: TProcess.WindowHeight (306), TProcess.WindowWidth (307), TProcess.WindowLeft (306), TProcess.WindowColumns (306), TProcess.WindowRows (307), TProcess.FillAttribute (308), TProcess.StartupOptions (304)

23.5.42 TProcess.WindowWidth

Synopsis: Height of the process main window (Windows only)

Declaration: `Property WindowWidth : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowWidth` is the initial width (in pixels) of the process' main window. This property is only effective if `suoUseSize` is specified in StartupOptions (304)

See also: TProcess.WindowHeight (306), TProcess.WindowLeft (306), TProcess.WindowTop (307), TProcess.WindowColumns (306), TProcess.WindowRows (307), TProcess.FillAttribute (308), TProcess.StartupOptions (304)

23.5.43 TProcess.FillAttribute

Synopsis: Color attributes of the characters in the console window (Windows only)

Declaration: `Property FillAttribute : Cardinal`

Visibility: published

Access: Read,Write

Description: `FillAttribute` is a WORD value which specifies the background and foreground colors of the console window.

See also: TProcess.WindowHeight (306), TProcess.WindowWidth (307), TProcess.WindowLeft (306), TProcess.WindowTop (307), TProcess.WindowColumns (306), TProcess.WindowRows (307), TProcess.StartupOptions (304)

Chapter 24

Reference for unit 'rttiutils'

24.1 Used units

Table 24.1: Used units by unit 'rttiutils'

Name	Page
Classes	??
StrUtils	309
sysutils	??
typinfo	??

24.2 Overview

The rttiutils unit is a unit providing simplified access to the RTTI information from published properties using the TPropInfoList (311) class. This access can be used when saving or restoring form properties at runtime, or for persisting other objects whose RTTI is available: the TPropsStorage (314) class can be used for this. The implementation is based on the apputils unit from `RXLib` by *AO ROSNO* and *Master-Bank*

24.3 Constants, types and variables

24.3.1 Constants

```
sPropNameDelimiter : String = '_'
```

Separator used when constructing section/key names

24.3.2 Types

```
TEraseSectEvent = procedure(const ASection: String) of object
```

TEraseSectEvent is used by TPropsStorage (314) to clear a storage section, in a .ini file like fashion: The call should remove all keys in the section ASection, and remove the section from storage.

```
TFindComponentEvent = function(const Name: String) : TComponent
```

TFindComponentEvent should return the component instance for the component with name path Name. The name path should be relative to the global list of loaded components.

```
TReadStrEvent = function(const ASection: String;const Item: String;
                         const Default: String) : String of object
```

TReadStrEvent is used by TPropsStorage (314) to read strings from a storage mechanism, in a .ini file like fashion: The call should read the string in ASection with key Item, and if it does not exist, Default should be returned.

```
TWriteStrEvent = procedure(const ASection: String;const Item: String;
                           const Value: String) of object
```

TWriteStrEvent is used by TPropsStorage (314) to write strings to a storage mechanism, in a .ini file like fashion: The call should write the string Value in ASection with key Item. The section and key should be created if they didn't exist yet.

24.3.3 Variables

```
FindGlobalComponentCallBack : TFindComponentEvent
```

FindGlobalComponentCallBack is called by UpdateStoredList (311) whenever it needs to resolve component references. It should be set to a routine that locates a loaded component in the global list of loaded components.

24.4 Procedures and functions

24.4.1 CreateStoredItem

Synopsis: Concatenates component and property name

Declaration:
```
function CreateStoredItem(const CompName: String;const PropName: String)
                          : String
```

Visibility: default

Description: CreateStoredItem concatenates CompName and PropName if they are both empty. The names are separated by a dot (.) character. If either of the names is empty, an empty string is returned.

This function can be used to create items for the list of properties such as used in UpdateStoredList (311), TPropsStorage.StoreObjectsProps (316) or TPropsStorage.LoadObjectsProps (315).

See also: ParseStoredItem (311), UpdateStoredList (311), TPropsStorage.StoreObjectsProps (316), TPropsStorage.LoadObjectsProps (315)

24.4.2 ParseStoredItem

Synopsis: Split a property reference to component reference and property name

Declaration:
```
function ParseStoredItem(const Item: String;var CompName: String;
                         var PropName: String) : Boolean
```

Visibility: default

Description: `ParseStoredItem` parses the property reference `Item` and splits it in a reference to a component (returned in `CompName`) and a name of a property (returned in `PropName`). This function basically does the opposite of CreateStoredItem (310). Note that both names should be non-empty, i.e., at least 1 dot character must appear in `Item`.

Errors: If an error occurred during parsing, `False` is returned.

See also: CreateStoredItem (310), UpdateStoredList (311), TPropsStorage.StoreObjectsProps (316), TPropsStorage.LoadObjectsProps (315)

24.4.3 UpdateStoredList

Synopsis: Update a stringlist with object references

Declaration:
```
procedure UpdateStoredList(AComponent: TComponent;AStoredList: TStrings;
                           FromForm: Boolean)
```

Visibility: default

Description: `UpdateStoredList` will parse the strings in `AStoredList` using ParseStoredItem (311) and will replace the `Objects` properties with the instance of the object whose name each property path in the list refers to. If `FromForm` is `True`, then all instances are searched relative to `AComponent`, i.e. they must be owned by `AComponent`. If `FromForm` is `False` the instances are searched in the global list of streamed components. (the FindGlobalComponentCallBack (310) callback must be set for the search to work correctly in this case)

If a component cannot be found, the reference string to the property is removed from the stringlist.

Errors: If `AComponent` is `Nil`, an exception may be raised.

See also: ParseStoredItem (311), TPropsStorage.StoreObjectsProps (316), TPropsStorage.LoadObjectsProps (315), FindGlobalComponentCallBack (310)

24.5 TPropInfoList

24.5.1 Description

`TPropInfoList` is a class which can be used to maintain a list with information about published properties of a class (or an instance). It is used internally by TPropsStorage (314)

24.5.2 Method overview

Page	Property	Description
312	Contains	Check whether a certain property is included
312	Create	Create a new instance of `TPropInfoList`
313	Delete	Delete property information from the list
312	Destroy	Remove the `TPropInfoList` instance from memory
312	Find	Retrieve property information based on name
313	Intersect	Intersect 2 property lists

24.5.3 Property overview

Page	Property	Access	Description
313	Count	r	Number of items in the list
313	Items	r	Indexed access to the property type pointers

24.5.4 TPropInfoList.Create

Synopsis: Create a new instance of `TPropInfoList`

Declaration: `constructor Create(AObject: TObject;Filter: TTypeKinds)`

Visibility: public

Description: `Create` allocates and initializes a new instance of `TPropInfoList` on the heap. It retrieves a list of published properties from `AObject`: if `Filter` is empty, then all properties are retrieved. If it is not empty, then only properties of the kind specified in the set are retrieved. `Instance` should not be `Nil`

See also: TPropInfoList.Destroy (312)

24.5.5 TPropInfoList.Destroy

Synopsis: Remove the `TPropInfoList` instance from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the internal structures maintained by `TPropInfoList` and then calls the inherited `Destroy`.

See also: TPropInfoList.Create (312)

24.5.6 TPropInfoList.Contains

Synopsis: Check whether a certain property is included

Declaration: `function Contains(P: PPropInfo) : Boolean`

Visibility: public

Description: `Contains` checks whether `P` is included in the list of properties, and returns `True` if it does. If `P` cannot be found, `False` is returned.

See also: TPropInfoList.Find (312), TPropInfoList.Intersect (313)

24.5.7 TPropInfoList.Find

Synopsis: Retrieve property information based on name

Declaration: `function Find(const AName: String) : PPropInfo`

Visibility: public

Description: `Find` returns a pointer to the type information of the property `AName`. If no such information is available, the function returns `Nil`. The search is performed case insensitive.

See also: TPropInfoList.Intersect (313), TPropInfoList.Contains (312)

24.5.8 TPropInfoList.Delete

Synopsis: Delete property information from the list

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` deletes the property information at position `Index` from the list. It's mainly of use in the Intersect (313) call.

Errors: No checking on the validity of `Index` is performed.

See also: TPropInfoList.Intersect (313)

24.5.9 TPropInfoList.Intersect

Synopsis: Intersect 2 property lists

Declaration: `procedure Intersect(List: TPropInfoList)`

Visibility: public

Description: `Intersect` reduces the list of properties to the ones also contained in `List`, i.e. all properties which are not also present in `List` are removed.

See also: TPropInfoList.Delete (313), TPropInfoList.Contains (312)

24.5.10 TPropInfoList.Count

Synopsis: Number of items in the list

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the number of property type pointers in the list.

See also: TPropInfoList.Items (313)

24.5.11 TPropInfoList.Items

Synopsis: Indexed access to the property type pointers

Declaration: `Property Items[Index: Integer]: PPropInfo; default`

Visibility: public

Access: Read

Description: `Items` provides access to the property type pointers stored in the list. `Index` runs from 0 to `Count-1`.

See also: TPropInfoList.Count (313)

24.6 TPropsStorage

24.6.1 Description

`TPropsStorage` provides a mechanism to store properties from any class which has published properties (usually a `TPersistent` descendent) in a storage mechanism.

`TPropsStorage` does not handle the storage by itself, instead, the storage is handled through a series of callbacks to read and/or write strings. Conversion of property types to string is handled by `TPropsStorage` itself: all that needs to be done is set the 3 handlers. The storage mechanism is assumed to have the structure of an .ini file : sections with key/value pairs. The three callbacks should take this into account, but they do not need to create an actual .ini file.

24.6.2 Method overview

Page	Property	Description
314	LoadAnyProperty	Load a property value
315	LoadObjectsProps	Load a list of component properties
315	LoadProperties	Load a list of properties
314	StoreAnyProperty	Store a property value
316	StoreObjectsProps	Store a list of component properties
315	StoreProperties	Store a list of properties

24.6.3 Property overview

Page	Property	Access	Description
317	AObject	rw	Object to load or store properties from
318	OnEraseSection	rw	Erase a section in storage
317	OnReadString	rw	Read a string value from storage
318	OnWriteString	rw	Write a string value to storage
317	Prefix	rw	Prefix to use in storage
317	Section	rw	Section name for storage

24.6.4 TPropsStorage.StoreAnyProperty

Synopsis: Store a property value

Declaration: `procedure StoreAnyProperty(PropInfo: PPropInfo)`

Visibility: public

Description: `StoreAnyProperty` stores the property with information specified in `PropInfo` in the storage mechanism. The property value is retrieved from the object instance specified in the AObject (317) property of `TPropsStorage`.

Errors: If the property pointer is invalid or `AObject` is invalid, an exception will be raised.

See also: TPropsStorage.AObject (317), TPropsStorage.LoadAnyProperty (314), TPropsStorage.LoadProperties (315), TPropsStorage.StoreProperties (315)

24.6.5 TPropsStorage.LoadAnyProperty

Synopsis: Load a property value

Declaration: `procedure LoadAnyProperty(PropInfo: PPropInfo)`

Visibility: public

Description: `LoadAnyProperty` loads the property with information specified in `PropInfo` from the storage mechanism. The value is then applied to the object instance specified in the AObject (317) property of `TPropsStorage`.

Errors: If the property pointer is invalid or `AObject` is invalid, an exception will be raised.

See also: TPropsStorage.AObject (317), TPropsStorage.StoreAnyProperty (314), TPropsStorage.LoadProperties (315), TPropsStorage.StoreProperties (315)

24.6.6 TPropsStorage.StoreProperties

Synopsis: Store a list of properties

Declaration: `procedure StoreProperties(PropList: TStrings)`

Visibility: public

Description: `StoreProperties` stores the values of all properties in `PropList` in the storage mechanism. The list should contain names of published properties of the AObject (317) object.

Errors: If an invalid property name is specified, an exception will be raised.

See also: TPropsStorage.AObject (317), TPropsStorage.StoreAnyProperty (314), TPropsStorage.LoadProperties (315), TPropsStorage.LoadAnyProperty (314)

24.6.7 TPropsStorage.LoadProperties

Synopsis: Load a list of properties

Declaration: `procedure LoadProperties(PropList: TStrings)`

Visibility: public

Description: `LoadProperties` loads the values of all properties in `PropList` from the storage mechanism. The list should contain names of published properties of the AObject (317) object.

Errors: If an invalid property name is specified, an exception will be raised.

See also: TPropsStorage.AObject (317), TPropsStorage.StoreAnyProperty (314), TPropsStorage.StoreProperties (315), TPropsStorage.LoadAnyProperty (314)

24.6.8 TPropsStorage.LoadObjectsProps

Synopsis: Load a list of component properties

Declaration: `procedure LoadObjectsProps(AComponent: TComponent;StoredList: TStrings)`

Visibility: public

Description: `LoadObjectsProps` loads a list of component properties, relative to `AComponent`: the names of the component properties to load are specified as follows:

```
ComponentName1.PropertyName
ComponentName2.Subcomponent1.PropertyName
```

The component instances will be located relative to `AComponent`, and must therefore be names of components owned by `AComponent`, followed by a valid property of these components. If the componentname is missing, the property name will be assumed to be a property of `AComponent` itself.

The `Objects` property of the stringlist should be filled with the instances of the components the property references refer to: they can be filled with the UpdateStoredList (311) call.

For example, to load the checked state of a checkbox named 'CBCheckMe' and the caption of a button named 'BPressMe', both owned by a form, the following strings should be passed:

```
CBCheckMe.Checked
BPressMe.Caption
```

and the `ACompontent` should be the form component that owns the button and checkbox.

Note that this call removes the value of the AObject (317) property.

Errors: If an invalid component is specified, an exception will be raised.

See also: UpdateStoredList (311), TPropsStorage.StoreObjectsProps (316), TPropsStorage.LoadProperties (315), TPropsStorage.LoadAnyProperty (314)

24.6.9 TPropsStorage.StoreObjectsProps

Synopsis: Store a list of component properties

Declaration: `procedure StoreObjectsProps(AComponent: TComponent;StoredList: TStrings)`

Visibility: public

Description: `StoreObjectsProps` stores a list of component properties, relative to `AComponent`: the names of the component properties to store are specified as follows:

```
ComponentName1.PropertyName
ComponentName2.Subcomponent1.PropertyName
```

The component instances will be located relative to `AComponent`, and must therefore be names of components owned by `AComponent`, followed by a valid property of these components. If the componentname is missing, the property name will be assumed to be a property of `AComponent` itself.

The `Objects` property of the stringlist should be filled with the instances of the components the property references refer to: they can be filled with the UpdateStoredList (311) call.

For example, to store the checked state of a checkbox named 'CBCheckMe' and the caption of a button named 'BPressMe', both owned by a form, the following strings should be passed:

```
CBCheckMe.Checked
BPressMe.Caption
```

and the `ACompontent` should be the form component that owns the button and checkbox.

Note that this call removes the value of the AObject (317) property.

See also: UpdateStoredList (311), TPropsStorage.LoadObjectsProps (315), TPropsStorage.LoadProperties (315), TPropsStorage.LoadAnyProperty (314)

24.6.10 TPropsStorage.AObject

Synopsis: Object to load or store properties from

Declaration: `Property AObject : TObject`

Visibility: public

Access: Read,Write

Description: `AObject` is the object instance whose properties will be loaded or stored with any of the methods in the `TPropsStorage` class. Note that a call to StoreObjectProps (316) or LoadObjectProps (315) will destroy any value that this property might have.

See also: TPropsStorage.LoadProperties (315), TPropsStorage.LoadAnyProperty (314), TPropsStorage.StoreProperties (315), TPropsStorage.StoreAnyProperty (314), TPropsStorage.StoreObjectsProps (316), TPropsStorage.LoadObjectsProps (315)

24.6.11 TPropsStorage.Prefix

Synopsis: Prefix to use in storage

Declaration: `Property Prefix : String`

Visibility: public

Access: Read,Write

Description: `Prefix` is prepended to all property names to form the key name when writing a property to storage, or when reading a value from storage. This is useful when storing properties of multiple forms in a single section.

See also: TPropsStorage.Section (317)

24.6.12 TPropsStorage.Section

Synopsis: Section name for storage

Declaration: `Property Section : String`

Visibility: public

Access: Read,Write

Description: `Section` is used as the section name when writing values to storage. Note that when writing properties of subcomponents, their names will be appended to the value specified here.

See also: TPropsStorage.Section (317)

24.6.13 TPropsStorage.OnReadString

Synopsis: Read a string value from storage

Declaration: `Property OnReadString : TReadStrEvent`

Visibility: public

Access: Read,Write

Description: `OnReadString` is the event handler called whenever `TPropsStorage` needs to read a string from storage. It should be set whenever properties need to be loaded, or an exception will be raised.

See also: TPropsStorage.OnWriteString (318), TPropsStorage.OnEraseSection (318), TReadStrEvent (310)

24.6.14 TPropsStorage.OnWriteString

Synopsis: Write a string value to storage

Declaration: `Property OnWriteString : TWriteStrEvent`

Visibility: public

Access: Read,Write

Description: `OnWriteString` is the event handler called whenever `TPropsStorage` needs to write a string to storage. It should be set whenever properties need to be stored, or an exception will be raised.

See also: TPropsStorage.OnReadString (317), TPropsStorage.OnEraseSection (318), TWriteStrEvent (310)

24.6.15 TPropsStorage.OnEraseSection

Synopsis: Erase a section in storage

Declaration: `Property OnEraseSection : TEraseSectEvent`

Visibility: public

Access: Read,Write

Description: `OnEraseSection` is the event handler called whenever `TPropsStorage` needs to clear a complete storage section. It should be set whenever stringlist properties need to be stored, or an exception will be raised.

See also: TPropsStorage.OnReadString (317), TPropsStorage.OnWriteString (318), TEraseSectEvent (309)

Chapter 25

Reference for unit 'simpleipc'

25.1 Used units

Table 25.1: Used units by unit 'simpleipc'

Name	Page
Classes	??
sysutils	??

25.2 Overview

The SimpleIPC unit provides classes to implement a simple, one-way IPC mechanism using string messages. It provides a TSimpleIPCServer (329) component for the server, and a TSimpleIPCClient (326) component for the client. The components are cross-platform, and should work both on Windows and unix-like systems.

25.3 Constants, types and variables

25.3.1 Resource strings

```
SErrActive = 'This operation is illegal when the server is active.'
```

Error message if client/server is active.

```
SErrInActive = 'This operation is illegal when the server is inactive.'
```

Error message if client/server is not active.

```
SErrServerNotActive = 'Server with ID %s is not active.'
```

Error message if server is not active

25.3.2 Constants

`MsgVersion = 1`

Current version of the messaging protocol

`mtString = 1`

String message type

`mtUnknown = 0`

Unknown message type

25.3.3 Types

`TIPCClientCommClass = Class of TIPCClientComm`

`TIPCClientCommClass` is used by TSimpleIPCClient (326) to decide which kind of communication channel to set up.

`TIPCServerCommClass = Class of TIPCServerComm`

`TIPCServerCommClass` is used by TSimpleIPCServer (329) to decide which kind of communication channel to set up.

`TMessageType = LongInt`

`TMessageType` is provided for backward compatibility with earlier versions of the simpleipc unit.

```
TMsgHeader = packed record
  Version : Byte;
  MsgType : TMessageType;
  MsgLen : Integer;
end
```

`TMsgHeader` is used internally by the IPC client and server components to transmit data. The `Version` field denotes the protocol version. The `MsgType` field denotes the type of data (`mtString` for string messages), and `MsgLen` is the length of the message which will follow.

25.3.4 Variables

`DefaultIPCClientClass : TIPCClientCommClass = nil`

`DefaultIPCClientClass` is filled with a class pointer indicating which kind of communication protocol class should be instantiated by the TSimpleIPCClient (326) class. It is set to a default value by the default implementation in the SimpleIPC unit, but can be set to another class if another method of transport is desired. (it should match the communication protocol used by the server, obviously).

`DefaultIPCServerClass : TIPCServerCommClass = nil`

`DefaultIPCServerClass` is filled with a class pointer indicating which kind of communication protocol class should be instantiated by the TSimpleIPCServer (329) class. It is set to a default value by the default implementation in the SimpleIPC unit, but can be set to another class if another method of transport is desired.

25.4 EIPCError

25.4.1 Description

`EIPCError` is the exception used by the various classes in the SimpleIPC unit to report errors.

25.5 TIPCClientComm

25.5.1 Description

`TIPCClientComm` is an abstract component which implements the client-side communication protocol. The behaviour expected of this class must be implemented in a platform-dependent descendent class.

The TSimpleIPCClient (326) class does not implement the messaging protocol by itself. Instead, it creates an instance of a (platform dependent) descendent of `TIPCClientComm` which handles the internals of the commnication protocol.

The server side of the messaging protocol is handled by the TIPCServerComm (323) component. The descenent components must always be implemented in pairs.

25.5.2 Method overview

Page	Property	Description
321	Connect	Connect to the server
321	Create	Create a new instance of the `TIPCClientComm`
322	Disconnect	Disconnect from the server
322	SendMessage	Send a message
322	ServerRunning	Check if the server is running.

25.5.3 Property overview

Page	Property	Access	Description
323	Owner	r	`TSimpleIPCClient` instance for which communication must be handled.

25.5.4 TIPCClientComm.Create

Synopsis: Create a new instance of the `TIPCClientComm`

Declaration: `constructor Create(AOwner: TSimpleIPCClient); Virtual`

Visibility: public

Description: `Create` instantiates a new instance of the `TIPCClientComm` class, and stores the `AOwner` reference to the TSimpleIPCClient (326) instance for which it will handle communitation. It can be retrieved later using the Owner (323) property.

See also: TIPCClientComm.Owner (323), TSimpleIPCClient (326)

25.5.5 TIPCClientComm.Connect

Synopsis: Connect to the server

Declaration: `procedure Connect; Virtual; Abstract`

Visibility: public

Description: `Connect` must establish a communication channel with the server. The server endpoint must be constructed from the ServerID (326) and ServerInstance (329) properties of the owning TSimpleIPCClient (326) instance.

`Connect` is called by the TSimpleIPCClient.Connect (327) call or when the Active (326) property is set to `True`

Messages can be sent only after `Connect` was called succesfully.

Errors: If the connection setup fails, or the connection was already set up , then an exception may be raised.

See also: TSimpleIPCClient.Connect (327), TSimpleIPC.Active (326), TIPCClientComm.Disconnect (322)

25.5.6 TIPCClientComm.Disconnect

Synopsis: Disconnect from the server

Declaration: `procedure Disconnect; Virtual; Abstract`

Visibility: public

Description: `Disconnect` closes the communication channel with the server. Any calls to `SendMessage` are invalid after `Disconnect` was called.

`Disconnect` is called by the TSimpleIPCClient.Disconnect (328) call or when the Active (326) property is set to `False`.

Messages can no longer be sent after `Disconnect` was called.

Errors: If the connection shutdown fails, or the connection was already shut down, then an exception may be raised.

See also: TSimpleIPCClient.Disconnect (328), TSimpleIPC.Active (326), TIPCClientComm.Connect (321)

25.5.7 TIPCClientComm.ServerRunning

Synopsis: Check if the server is running.

Declaration: `function ServerRunning : Boolean; Virtual; Abstract`

Visibility: public

Description: `ServerRunning` returns `True` if the server endpoint of the communication channel can be found, or `False` if not. The server endpoint should be obtained from the `ServerID` and `InstanceID` properties of the owning TSimpleIPCClient (326) component.

See also: TSimpleIPCClient.InstanceID (326), TSimpleIPCClient.ServerID (326)

25.5.8 TIPCClientComm.SendMessage

Synopsis: Send a message

Declaration: `procedure SendMessage(MsgType: TMessageType;Stream: TStream); Virtual ; Abstract`

Visibility: public

Description: `SendMessage` should deliver the message with type `MsgType` and data in `Stream` to the server. It should not return until the message was delivered.

Errors: If the delivery of the message fails, an exception will be raised.

25.5.9 TIPCClientComm.Owner

Synopsis: `TSimpleIPCClient` instance for which communication must be handled.

Declaration: `Property Owner : TSimpleIPCClient`

Visibility: public

Access: Read

Description: `Owner` is the TSimpleIPCClient (326) instance for which the communication must be handled. It cannot be changed, and must be specified when the `TIPCClientComm` instance is created.

See also: TSimpleIPCClient (326), TIPCClientComm.Create (321)

25.6 TIPCServerComm

25.6.1 Description

`TIPCServerComm` is an abstract component which implements the server-side communication protocol. The behaviour expected of this class must be implemented in a platform-dependent descendent class.

The TSimpleIPCServer (329) class does not implement the messaging protocol by itself. Instead, it creates an instance of a (platform dependent) descendent of `TIPCServerComm` which handles the internals of the commnication protocol.

The client side of the messaging protocol is handled by the TIPCClientComm (321) component. The descenent components must always be implemented in pairs.

25.6.2 Method overview

Page	Property	Description
323	Create	Create a new instance of the communication handler
324	PeekMessage	See if a message is available.
325	ReadMessage	Read message from the channel.
324	StartServer	Start the server-side of the communication channel
324	StopServer	Stop the server side of the communication channel.

25.6.3 Property overview

Page	Property	Access	Description
325	InstanceID	r	Unique identifier for the communication channel.
325	Owner	r	`TSimpleIPCServer` instance for which to handle transport

25.6.4 TIPCServerComm.Create

Synopsis: Create a new instance of the communication handler

Declaration: `constructor Create(AOwner: TSimpleIPCServer); Virtual`

Visibility: public

Description: `Create` initializes a new instance of the communication handler. It simply saves the `AOwner` parameter in the Owner (325) property.

See also: TIPCServerComm.Owner (325)

25.6.5 TIPCServerComm.StartServer

Synopsis: Start the server-side of the communication channel

Declaration: `procedure StartServer; Virtual; Abstract`

Visibility: public

Description: `StartServer` sets up the server-side of the communication channel. After `StartServer` was called, a client can connect to the communication channel, and send messages to the server.

It is called when the TSimpleIPC.Active (326) property of the TSimpleIPCServer (329) instance is set to `True`.

Errors: In case of an error, an EIPCError (321) exception is raised.

See also: TSimpleIPCServer (329), TSimpleIPC.Active (326)

25.6.6 TIPCServerComm.StopServer

Synopsis: Stop the server side of the communication channel.

Declaration: `procedure StopServer; Virtual; Abstract`

Visibility: public

Description: `StartServer` closes down the server-side of the communication channel. After `StartServer` was called, a client can no longer connect to the communication channel, or even send messages to the server if it was previously connected (i.e. it will be disconnected).

It is called when the TSimpleIPC.Active (326) property of the TSimpleIPCServer (329) instance is set to `False`.

Errors: In case of an error, an EIPCError (321) exception is raised.

See also: TSimpleIPCServer (329), TSimpleIPC.Active (326)

25.6.7 TIPCServerComm.PeekMessage

Synopsis: See if a message is available.

Declaration: `function PeekMessage(TimeOut: Integer) : Boolean; Virtual; Abstract`

Visibility: public

Description: `PeekMessage` can be used to see if a message is available: it returns `True` if a message is available. It will wait maximum `TimeOut` milliseconds for a message to arrive. If no message was available after this time, it will return `False`.

If a message was available, it can be read with the ReadMessage (325) call.

See also: TIPCServerComm.ReadMessage (325)

25.6.8 TIPCServerComm.ReadMessage

Synopsis: Read message from the channel.

Declaration: `procedure ReadMessage; Virtual; Abstract`

Visibility: public

Description: `ReadMessage` reads the message for the channel, and stores the information in the data structures in the `Owner` class.

`ReadMessage` is a blocking call: if no message is available, the program will wait till a message arrives. Use PeekMessage (324) to see if a message is available.

See also: TSimpleIPCServer (329)

25.6.9 TIPCServerComm.Owner

Synopsis: `TSimpleIPCServer` instance for which to handle transport

Declaration: `Property Owner : TSimpleIPCServer`

Visibility: public

Access: Read

Description: `Owner` refers to the TSimpleIPCServer (329) instance for which this instance of `TSimpleIPCServer` handles the transport. It is specified when the `TIPCServerComm` is created.

See also: TSimpleIPCServer (329)

25.6.10 TIPCServerComm.InstanceID

Synopsis: Unique identifier for the communication channel.

Declaration: `Property InstanceID : String`

Visibility: public

Access: Read

Description: `InstanceID` returns a textual representation which uniquely identifies the communication channel on the server. The value is system dependent, and should be usable by the client-side to establish a communication channel with this instance.

25.7 TSimpleIPC

25.7.1 Description

`TSimpleIPC` is the common ancestor for the TSimpleIPCServer (329) and TSimpleIPCClient (326) classes. It implements some common properties between client and server.

25.7.2 Property overview

Page	Property	Access	Description
326	Active	rw	Communication channel active
326	ServerID	rw	Unique server identification

25.7.3 TSimpleIPC.Active

Synopsis: Communication channel active

Declaration: `Property Active : Boolean`

Visibility: published

Access: Read,Write

Description: `Active` can be set to `True` to set up the client or server end of the communication channel. For the server this means that the server end is set up, for the client it means that the client tries to connect to the server with ServerID (326) identification.

See also: TSimpleIPC.ServerID (326)

25.7.4 TSimpleIPC.ServerID

Synopsis: Unique server identification

Declaration: `Property ServerID : String`

Visibility: published

Access: Read,Write

Description: `ServerID` is the unique server identification: on the server, it determines how the server channel is set up, on the client it determines the server with which to connect.

See also: TSimpleIPC.Active (326)

25.8 TSimpleIPCClient

25.8.1 Description

`TSimpleIPCClient` is the client side of the simple IPC communication protocol. The client program should create a `TSimpleIPCClient` instance, set its ServerID (326) property to the unique name for the server it wants to send messages to, and then set the Active (326) property to `True` (or call Connect (326)).

After the connection with the server was established, messages can be sent to the server with the SendMessage (328) or SendStringMessage (328) calls.

25.8.2 Method overview

Page	Property	Description
327	Connect	Connect to the server
327	Create	Create a new instance of `TSimpleIPCClient`
327	Destroy	Remove the `TSimpleIPCClient` instance from memory
328	Disconnect	Disconnect from the server
328	SendMessage	Send a message to the server
328	SendStringMessage	Send a string message to the server
329	SendStringMessageFmt	Send a formatted string message
328	ServerRunning	Check if the server is running.

25.8.3 Property overview

Page	Property	Access	Description
329	ServerInstance	rw	Server instance identification

25.8.4 TSimpleIPCClient.Create

Synopsis: Create a new instance of `TSimpleIPCClient`

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` instantiates a new instance of the `TSimpleIPCClient` class. It initializes the data structures needed to handle the client side of the communication.

See also: TSimpleIPCClient.Destroy (327)

25.8.5 TSimpleIPCClient.Destroy

Synopsis: Remove the `TSimpleIPCClient` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` disconnects the client from the server if need be, and cleans up the internal data structures maintained by `TSimpleIPCClient` and then calls the inherited `Destroy`, which will remove the instance from memory.

Never call `Destroy` directly, use the `Free` method instead or the `FreeAndNil` procedure in SysUtils.

See also: TSimpleIPCClient.Create (327)

25.8.6 TSimpleIPCClient.Connect

Synopsis: Connect to the server

Declaration: `procedure Connect`

Visibility: public

Description: `Connect` connects to the server indicated in the ServerID (326) and InstanceID (326) properties. `Connect` is called automatically if the Active (326) property is set to `True`.

After a successful call to `Connect`, messages can be sent to the server using SendMessage (328) or SendStringMessage (328).

Calling `Connect` if the connection is already open has no effect.

Errors: If creating the connection fails, an EIPCError (321) exception may be raised.

See also: TSimpleIPC.ServerID (326), TSimpleIPCClient.InstanceID (326), TSimpleIPC.Active (326), TSimpleIPCClient.SendMessage (328), TSimpleIPCClient.SendStringMessage (328), TSimpleIPCClient.Disconnect (328)

25.8.7 TSimpleIPCClient.Disconnect

Synopsis: Disconnect from the server

Declaration: `procedure Disconnect`

Visibility: public

Description: `Disconnect` shuts down the connection with the server as previously set up with Connect (327). `Disconnect` is called automatically if the Active (326) property is set to `False`.

After a successful call to `Disconnect`, messages can no longer be sent to the server. Attempting to do so will result in an exception.

Calling `Disconnect` if there is no connection has no effect.

Errors: If creating the connection fails, an EIPCError (321) exception may be raised.

See also: TSimpleIPC.Active (326), TSimpleIPCClient.Connect (327)

25.8.8 TSimpleIPCClient.ServerRunning

Synopsis: Check if the server is running.

Declaration: `function ServerRunning : Boolean`

Visibility: public

Description: `ServerRunning` verifies if the server indicated in the ServerID (326) and InstanceID (326) properties is running. It returns `True` if the server communication endpoint can be reached, `False` otherwise. This function can be called before a connection is made.

See also: TSimpleIPCClient.Connect (327)

25.8.9 TSimpleIPCClient.SendMessage

Synopsis: Send a message to the server

Declaration: `procedure SendMessage(MsgType: TMessageType;Stream: TStream)`

Visibility: public

Description: `SendMessage` sends a message of type `MsgType` and data from `stream` to the server. The client must be connected for this call to work.

Errors: In case an error occurs, or there is no connection to the server, an EIPCError (321) exception is raised.

See also: TSimpleIPCClient.Connect (327), TSimpleIPCClient.SendStringMessage (328)

25.8.10 TSimpleIPCClient.SendStringMessage

Synopsis: Send a string message to the server

Declaration:
```
procedure SendStringMessage(const Msg: String)
procedure SendStringMessage(MsgType: TMessageType;const Msg: String)
```

Visibility: public

Description: `SendStringMessage` sends a string message with type `MsgTyp` and data `Msg` to the server. This is a convenience function: a small wrapper around the SendMessage (328) method

Errors: Same as for `SendMessage`.

See also: TSimpleIPCClient.SendMessage (328), TSimpleIPCClient.Connect (327), TSimpleIPCClient.SendStringMessageFmt (329)

25.8.11 TSimpleIPCClient.SendStringMessageFmt

Synopsis: Send a formatted string message

Declaration:
```
procedure SendStringMessageFmt(const Msg: String;Args: Array of const)
procedure SendStringMessageFmt(MsgType: TMessageType;const Msg: String;
                               Args: Array of const)
```

Visibility: public

Description: `SendStringMessageFmt` sends a string message with type `MsgTyp` and message formatted from `Msg` and `Args` to the server. This is a convenience function: a small wrapper around the SendStringMessage (328) method

Errors: Same as for `SendMessage`.

See also: TSimpleIPCClient.SendMessage (328), TSimpleIPCClient.Connect (327), TSimpleIPCClient.SendStringMessage (328)

25.8.12 TSimpleIPCClient.ServerInstance

Synopsis: Server instance identification

Declaration: `Property ServerInstance : String`

Visibility: public

Access: Read,Write

Description: `ServerInstance` should be used in case a particular instance of the server identified with `ServerID` should be contacted. This must be used if the server has its GLobal (333) property set to `False`, and should match the server's InstanceID (332) property.

See also: TSimpleIPC.ServerID (326), TSimpleIPCServer.Global (333), TSimpleIPCServer.InstanceID (332)

25.9 TSimpleIPCServer

25.9.1 Description

`TSimpleIPCServer` is the server side of the simple IPC communication protocol. The server program should create a `TSimpleIPCServer` instance, set its ServerID (326) property to a unique name for the system, and then set the Active (326) property to `True` (or call StartServer (330)).

After the server was started, it can check for availability of messages with the PeekMessage (331) call, and read the message with ReadMessage (329).

25.9.2 Method overview

Page	Property	Description
330	Create	Create a new instance of `TSimpleIPCServer`
330	Destroy	Remove the `TSimpleIPCServer` instance from memory
331	GetMessageData	Read the data of the last message in a stream
331	PeekMessage	Check if a client message is available.
330	StartServer	Start the server
331	StopServer	Stop the server

25.9.3 Property overview

Page	Property	Access	Description
333	Global	rw	Is the server reachable to all users or not
332	InstanceID	r	Instance ID
332	MsgData	r	Last message data
332	MsgType	r	Last message type
333	OnMessage	rw	Event triggered when a pessage arrives
332	StringMessage	r	Last message as a string.

25.9.4 TSimpleIPCServer.Create

Synopsis: Create a new instance of `TSimpleIPCServer`

Declaration: `constructor Create(AOwner: TComponent);  Override`

Visibility: public

Description: `Create` instantiates a new instance of the `TSimpleIPCServer` class. It initializes the data structures needed to handle the server side of the communication.

See also: TSimpleIPCServer.Destroy (330)

25.9.5 TSimpleIPCServer.Destroy

Synopsis: Remove the `TSimpleIPCServer` instance from memory

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` stops the server, cleans up the internal data structures maintained by `TSimpleIPCServer` and then calls the inherited `Destroy`, which will remove the instance from memory.

Never call `Destroy` directly, use the `Free` method instead or the `FreeAndNil` procedure in SysUtils.

See also: TSimpleIPCServer.Create (330)

25.9.6 TSimpleIPCServer.StartServer

Synopsis: Start the server

Declaration: `procedure StartServer`

Visibility: public

Description: `StartServer` starts the server side of the communication channel. It is called automatically when the `Active` property is set to `True`. It creates the internal communication object (a TIPCServerComm (323) descendent) and activates the communication channel.

After this method was called, clients can connect and send messages.

Prior to calling this method, the ServerID (326) property must be set.

Errors: If an error occurs a EIPCError (321) exception may be raised.

See also: TIPCServerComm (323), TSimpleIPC.Active (326), TSimpleIPC.ServerID (326), TSimpleIPCServer.StopServer (331)

25.9.7 TSimpleIPCServer.StopServer

Synopsis: Stop the server

Declaration: `procedure StopServer`

Visibility: public

Description: `StopServer` stops the server side of the communication channel. It is called automatically when the `Active` property is set to `False`. It deactivates the communication channel and frees the internal communication object (a TIPCServerComm (323) descendent).

See also: TIPCServerComm (323), TSimpleIPC.Active (326), TSimpleIPC.ServerID (326), TSimpleIPCServer.StartServer (330)

25.9.8 TSimpleIPCServer.PeekMessage

Synopsis: Check if a client message is available.

Declaration: `function PeekMessage(TimeOut: Integer;DoReadMessage: Boolean) : Boolean`

Visibility: public

Description: `PeekMessage` checks if a message from a client is available. It will return `True` if a message is available. The call will wait for `TimeOut` milliseconds for a message to arrive: if after `TimeOut` milliseconds, no message is available, the function will return `False`.

If `DoReadMessage` is `True` then `PeekMessage` will read the message. If it is `False`, it does not read the message. The message should then be read manually with ReadMessage (329).

See also: TSimpleIPCServer.ReadMessage (329)

25.9.9 TSimpleIPCServer.GetMessageData

Synopsis: Read the data of the last message in a stream

Declaration: `procedure GetMessageData(Stream: TStream)`

Visibility: public

Description: `GetMessageData` reads the data of the last message from TSimpleIPCServer.MsgData (332) and stores it in stream `Stream`. If no data was available, the stream will be cleared.

This function will return valid data only after a succesful call to ReadMessage (329). It will also not clear the data buffer.

See also: TSimpleIPCServer.StringMessage (332), TSimpleIPCServer.MsgData (332), TSimpleIPCServer.MsgType (332)

25.9.10 TSimpleIPCServer.StringMessage

Synopsis: Last message as a string.

Declaration: `Property StringMessage : String`

Visibility: public

Access: Read

Description: `StringMessage` is the content of the last message as a string.

This property will contain valid data only after a succesful call to ReadMessage (329).

See also: TSimpleIPCServer.GetMessageData (331)

25.9.11 TSimpleIPCServer.MsgType

Synopsis: Last message type

Declaration: `Property MsgType : TMessageType`

Visibility: public

Access: Read

Description: `MsgType` contains the message type of the last message.

This property will contain valid data only after a succesful call to ReadMessage (329).

See also: TSimpleIPCServer.ReadMessage (329)

25.9.12 TSimpleIPCServer.MsgData

Synopsis: Last message data

Declaration: `Property MsgData : TStream`

Visibility: public

Access: Read

Description: `MsgData` contains the actual data from the last read message. If the data is a string, then StringMessage (332) is better suited to read the data.

This property will contain valid data only after a succesful call to ReadMessage (329).

See also: TSimpleIPCServer.StringMessage (332), TSimpleIPCServer.ReadMessage (329)

25.9.13 TSimpleIPCServer.InstanceID

Synopsis: Instance ID

Declaration: `Property InstanceID : String`

Visibility: public

Access: Read

Description: `InstanceID` is the unique identifier for this server communication channel endpoint, and will be appended to the ServerID (329) property to form the unique server endpoint which a client should use.

See also: TSimpleIPCServer.ServerID (329), TSimpleIPCServer.GlobalID (329)

25.9.14 TSimpleIPCServer.Global

Synopsis: Is the server reachable to all users or not

Declaration: `Property Global : Boolean`

Visibility: published

Access: Read,Write

Description: `Global` indicates whether the server is reachable to all users (`True`) or if it is private to the current process (`False`). In the latter case, the unique channel endpoint identification may change: a unique identification of the current process is appended to the `ServerID` name.

See also: TSimpleIPCServer.ServerID (329), TSimpleIPCServer.InstanceID (332)

25.9.15 TSimpleIPCServer.OnMessage

Synopsis: Event triggered when a pessage arrives

Declaration: `Property OnMessage : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnMessage` is called by ReadMessage (329) when a message has been read. The actual message data can be retrieved with one of the StringMessage (332), MsgData (332) or MsgType (332) properties.

See also: TSimpleIPCServer.StringMessage (332), TSimpleIPCServer.MsgData (332), TSimpleIPCServer.MsgType (332)

Chapter 26

Reference for unit 'streamcoll'

26.1 Used units

Table 26.1: Used units by unit 'streamcoll'

Name	Page
Classes	??
sysutils	??

26.2 Overview

The streamcoll unit contains the implentation of a collection (and corresponding collection item) which implements routines for saving or loading the collection to/from a stream. The collection item should implement 2 routines to implement the streaming; the streaming itself is not performed by the TStreamCollection (337) collection item.

The streaming performed here is not compatible with the streaming implemented in the `Classes` unit for components. It is independent of the latter and can be used without a component to hold the collection.

The collection item introduces mostly protected methods, and the unit contains a lot of auxiliary routines which aid in streaming.

26.3 Procedures and functions

26.3.1 ColReadBoolean

Synopsis: Read a boolean value from a stream

Declaration: `function ColReadBoolean(S: TStream) : Boolean`

Visibility: default

Description: `ColReadBoolean` reads a boolean from the stream `S` as it was written by ColWriteBoolean (336) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: ColReadDateTime (335), ColWriteBoolean (336), ColReadString (336), ColReadInteger (335), ColReadFloat (335), ColReadCurrency (335)

26.3.2 ColReadCurrency

Synopsis: Read a currency value from the stream

Declaration: `function ColReadCurrency(S: TStream) : Currency`

Visibility: default

Description: `ColReadCurrency` reads a currency value from the stream `S` as it was written by ColWriteCurrency (336) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: ColReadDateTime (335), ColReadBoolean (334), ColReadString (336), ColReadInteger (335), ColReadFloat (335), ColWriteCurrency (336)

26.3.3 ColReadDateTime

Synopsis: Read a `TDateTime` value from a stream

Declaration: `function ColReadDateTime(S: TStream) : TDateTime`

Visibility: default

Description: `ColReadDateTime` reads a currency value from the stream `S` as it was written by ColWriteDateTime (336) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: ColWriteDateTime (336), ColReadBoolean (334), ColReadString (336), ColReadInteger (335), ColReadFloat (335), ColReadCurrency (335)

26.3.4 ColReadFloat

Synopsis: Read a floating point value from a stream

Declaration: `function ColReadFloat(S: TStream) : Double`

Visibility: default

Description: `ColReadFloat` reads a double value from the stream `S` as it was written by ColWriteFloat (337) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: ColReadDateTime (335), ColReadBoolean (334), ColReadString (336), ColReadInteger (335), ColWriteFloat (337), ColReadCurrency (335)

26.3.5 ColReadInteger

Synopsis: Read a 32-bit integer from a stream.

Declaration: `function ColReadInteger(S: TStream) : Integer`

Visibility: default

Description: `ColReadInteger` reads a 32-bit integer from the stream `S` as it was written by ColWriteInteger (337) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: ColReadDateTime (335), ColReadBoolean (334), ColReadString (336), ColWriteInteger (337), ColReadFloat (335), ColReadCurrency (335)

26.3.6 ColReadString

Synopsis: Read a string from a stream

Declaration: `function ColReadString(S: TStream) : String`

Visibility: default

Description: `ColReadStream` reads a string value from the stream `S` as it was written by ColWriteString (337) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: ColReadDateTime (335), ColReadBoolean (334), ColWriteString (337), ColReadInteger (335), ColReadFloat (335), ColReadCurrency (335)

26.3.7 ColWriteBoolean

Synopsis: Write a boolean to a stream

Declaration: `procedure ColWriteBoolean(S: TStream;AValue: Boolean)`

Visibility: default

Description: `ColWriteBoolean` writes the boolean `AValue` to the stream. `S`.

See also: ColReadBoolean (334), ColWriteString (337), ColWriteInteger (337), ColWriteCurrency (336), ColWriteDateTime (336), ColWriteFloat (337)

26.3.8 ColWriteCurrency

Synopsis: Write a currency value to stream

Declaration: `procedure ColWriteCurrency(S: TStream;AValue: Currency)`

Visibility: default

Description: `ColWriteCurrency` writes the currency `AValue` to the stream `S`.

See also: ColWriteBoolean (336), ColWriteString (337), ColWriteInteger (337), ColWriteDateTime (336), ColWriteFloat (337), ColReadCurrency (335)

26.3.9 ColWriteDateTime

Synopsis: Write a `TDateTime` value to stream

Declaration: `procedure ColWriteDateTime(S: TStream;AValue: TDateTime)`

Visibility: default

Description: `ColWriteDateTime` writes the `TDateTimeAValue` to the stream `S`.

See also: ColReadDateTime (335), ColWriteBoolean (336), ColWriteString (337), ColWriteInteger (337), ColWriteFloat (337), ColWriteCurrency (336)

26.3.10 ColWriteFloat

Synopsis: Write floating point value to stream

Declaration: `procedure ColWriteFloat(S: TStream;AValue: Double)`

Visibility: default

Description: `ColWriteFloat` writes the double `AValue` to the stream `S`.

See also: ColWriteDateTime (336), ColWriteBoolean (336), ColWriteString (337), ColWriteInteger (337), ColReadFloat (335), ColWriteCurrency (336)

26.3.11 ColWriteInteger

Synopsis: Write a 32-bit integer to a stream

Declaration: `procedure ColWriteInteger(S: TStream;AValue: Integer)`

Visibility: default

Description: `ColWriteInteger` writes the 32-bit integer `AValue` to the stream `S`. No endianness is observed.

See also: ColWriteBoolean (336), ColWriteString (337), ColReadInteger (335), ColWriteCurrency (336), ColWriteDateTime (336)

26.3.12 ColWriteString

Synopsis: Write a string value to the stream

Declaration: `procedure ColWriteString(S: TStream;AValue: String)`

Visibility: default

Description: `ColWriteString` writes the string value `AValue` to the stream `S`.

See also: ColWriteBoolean (336), ColReadString (336), ColWriteInteger (337), ColWriteCurrency (336), ColWriteDateTime (336), ColWriteFloat (337)

26.4 EStreamColl

26.4.1 Description

Exception raised when an error occurs when streaming the collection.

26.5 TStreamCollection

26.5.1 Description

`TStreamCollection` is a TCollection (**??**) descendent which implements 2 calls LoadFromStream (338) and SaveToStream (338) which load and save the contents of the collection to a stream.

The collection items must be descendents of the TStreamCollectionItem (339) class for the streaming to work correctly.

Note that the stream must be used to load collections of the same type.

26.5.2 Method overview

Page	Property	Description
338	LoadFromStream	Load the collection from a stream
338	SaveToStream	Load the collection from the stream.

26.5.3 Property overview

Page	Property	Access	Description
338	Streaming	r	Indicates whether the collection is currently being written to stream

26.5.4 TStreamCollection.LoadFromStream

Synopsis: Load the collection from a stream

Declaration: `procedure LoadFromStream(S: TStream)`

Visibility: public

Description: `LoadFromStream` loads the collection from the stream `S`, if the collection was saved using SaveToStream (338). It reads the number of items in the collection, and then creates and loads the items one by one from the stream.

Errors: An exception may be raised if the stream contains invalid data.

See also: TStreamCollection.SaveToStream (338)

26.5.5 TStreamCollection.SaveToStream

Synopsis: Load the collection from the stream.

Declaration: `procedure SaveToStream(S: TStream)`

Visibility: public

Description: `SaveToStream` saves the collection to the stream `S` so it can be read from the stream with LoadFromStream (338). It does this by writing the number of collection items to the stream, and then streaming all items in the collection by calling their `SaveToStream` method.

Errors: None.

See also: TStreamCollection.LoadFromStream (338)

26.5.6 TStreamCollection.Streaming

Synopsis: Indicates whether the collection is currently being written to stream

Declaration: `Property Streaming : Boolean`

Visibility: public

Access: Read

Description: `Streaming` is set to `True` if the collection is written to or loaded from stream, and is set again to `False` if the streaming process is finished.

See also: TStreamCollection.LoadFromStream (338), TStreamCollection.SaveToStream (338)

26.6 TStreamCollectionItem

26.6.1 Description

`TStreamCollectionItem` is a TCollectionItem (**??**) descendent which implements 2 abstract routines: `LoadFromStream` and `SaveToStream` which must be overridden in a descendent class.

These 2 routines will be called by the TStreamCollection (337) to save or load the item from the stream.

Chapter 27

Reference for unit 'streamex'

27.1 Used units

Table 27.1: Used units by unit 'streamex'

Name	Page
Classes	**??**

27.2 Overview

streamex implements some extensions to be used together with streams from the classes unit.

27.3 TBidirBinaryObjectReader

27.3.1 Description

`TBidirBinaryObjectReader` is a class descendent from TBinaryObjectReader (**??**), which implements the necessary support for BiDi data: the position in the stream (not available in the standard streaming) is emulated.

27.3.2 Property overview

Page	Property	Access	Description
340	Position	rw	Position in the stream

27.3.3 TBidirBinaryObjectReader.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read,Write

Description: `Position` exposes the position of the stream in the reader for use in the TDelphiReader (341) class.

See also: TDelphiReader (341)

27.4 TBidirBinaryObjectWriter

27.4.1 Description

`TBidirBinaryObjectReader` is a class descendent from TBinaryObjectWriter (**??**), which implements the necessary support for BiDi data.

27.4.2 Property overview

Page	Property	Access	Description
341	Position	rw	Position in the stream

27.4.3 TBidirBinaryObjectWriter.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read,Write

Description: `Position` exposes the position of the stream in the writer for use in the TDelphiWriter (342) class.

See also: TDelphiWriter (342)

27.5 TDelphiReader

27.5.1 Description

`TDelphiReader` is a descendent of `TReader` which has support for BiDi Streaming. It overrides the stream reading methods for strings, and makes sure the stream can be positioned in the case of strings. For this purpose, it makes use of the TBidirBinaryObjectReader (340) driver class.

27.5.2 Method overview

Page	Property	Description
342	GetDriver	Return the driver class as a TBidirBinaryObjectReader (340) class
342	Read	Read data from stream
342	ReadStr	Overrides the standard `ReadStr` method

27.5.3 Property overview

Page	Property	Access	Description
342	Position	rw	Position in the stream

27.5.4 TDelphiReader.GetDriver

Synopsis: Return the driver class as a TBidirBinaryObjectReader (340) class

Declaration: `function GetDriver : TBidirBinaryObjectReader`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as TBidirBinaryObjectReader (340) class.

See also: TBidirBinaryObjectReader (340)

27.5.5 TDelphiReader.ReadStr

Synopsis: Overrides the standard `ReadStr` method

Declaration: `function ReadStr : String`

Visibility: public

Description: `ReadStr` makes sure the TBidirBinaryObjectReader (340) methods are used, to store additional information about the stream position when reading the strings.

See also: TBidirBinaryObjectReader (340)

27.5.6 TDelphiReader.Read

Synopsis: Read data from stream

Declaration: `procedure Read(var Buf;Count: LongInt);  Override`

Visibility: public

Description: `Read` reads raw data from the stream. It reads `Count` bytes from the stream and places them in `Buf`. It forces the use of the TBidirBinaryObjectReader (340) class when reading.

See also: TBidirBinaryObjectReader (340), TDelphiReader.Position (342)

27.5.7 TDelphiReader.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read,Write

Description: Position in the stream.

See also: TDelphiReader.Read (342)

27.6 TDelphiWriter

27.6.1 Description

`TDelphiWriter` is a descendent of `TWriter` which has support for BiDi Streaming. It overrides the stream writing methods for strings, and makes sure the stream can be positioned in the case of strings. For this purpose, it makes use of the TBidirBinaryObjectWriter (341) driver class.

27.6.2 Method overview

Page	Property	Description
343	FlushBuffer	Flushes the stream buffer
343	GetDriver	Return the driver class as a TBidirBinaryObjectWriter (341) class
343	Write	Write raw data to the stream
343	WriteStr	Write a string to the stream
344	WriteValue	Write value type

27.6.3 Property overview

Page	Property	Access	Description
344	Position	rw	Position in the stream

27.6.4 TDelphiWriter.GetDriver

Synopsis: Return the driver class as a TBidirBinaryObjectWriter (341) class

Declaration: `function GetDriver : TBidirBinaryObjectWriter`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as TBidirBinaryObjectWriter (341) class.

See also: TBidirBinaryObjectWriter (341)

27.6.5 TDelphiWriter.FlushBuffer

Synopsis: Flushes the stream buffer

Declaration: `procedure FlushBuffer`

Visibility: public

Description: `FlushBuffer` flushes the internal buffer of the writer. It simply calls the `FlushBuffer` method of the driver class.

27.6.6 TDelphiWriter.Write

Synopsis: Write raw data to the stream

Declaration: `procedure Write(const Buf;Count: LongInt);  Override`

Visibility: public

Description: `Write` writes `Count` bytes from `Buf` to the buffer, updating the position as needed.

27.6.7 TDelphiWriter.WriteStr

Synopsis: Write a string to the stream

Declaration: `procedure WriteStr(const Value: String)`

Visibility: public

Description: `WriteStr` writes a string to the stream, forcing the use of the TBidirBinaryObjectWriter (341) class methods, which update the position of the stream.

See also: TBidirBinaryObjectWriter (341)

27.6.8 TDelphiWriter.WriteValue

Synopsis: Write value type

Declaration: `procedure WriteValue(Value: TValueType)`

Visibility: public

Description: `WriteValue` overrides the same method in `TWriter` to force the use of the TBidirBinaryObjectWriter (341) methods, which update the position of the stream.

See also: TBidirBinaryObjectWriter (341)

27.6.9 TDelphiWriter.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read,Write

Description: `Position` exposes the position in the stream as exposed by the TBidirBinaryObjectWriter (341) instance used when streaming.

See also: TBidirBinaryObjectWriter (341)

Chapter 28

Reference for unit 'StreamIO'

28.1 Used units

Table 28.1: Used units by unit 'StreamIO'

Name	Page
Classes	**??**
sysutils	**??**

28.2 Overview

The StreamIO unit implements a call to reroute the input or output of a text file to a descendents of TStream (**??**).

This allows to use the standard pascal Read (**??**) and Write (**??**) functions (with all their possibilities), on streams.

28.3 Procedures and functions

28.3.1 AssignStream

Synopsis: Assign a text file to a stream.

Declaration: `procedure AssignStream(var F: Textfile;Stream: TStream)`

Visibility: default

Description: `AssignStream` assigns the stream `Stream` to file `F`. The file can subsequently be used to write to the stream, using the standard Write (**??**) calls.

Before writing, call Rewrite (**??**) on the stream. Before reading, call Reset (**??**).

Errors: if `Stream` is `Nil`, an exception will be raised.

See also: #rtl.classes.TStream (**??**), GetStream (346)

28.3.2 GetStream

Synopsis: Return the stream, associated with a file.

Declaration: `function GetStream(var F: TTextRec) : TStream`

Visibility: default

Description: `GetStream` returns the instance of the stream that was associated with the file `F` using AssignStream (345).

Errors: An invalid class reference will be returned if the file was not associated with a stream.

See also: AssignStream (345), #rtl.classes.TStream (**??**)

Chapter 29

Reference for unit 'syncobjs'

29.1 Used units

Table 29.1: Used units by unit 'syncobjs'

Name	Page
sysutils	??

29.2 Overview

The syncobjs unit implements some classes which can be used when synchronizing threads in routines or classes that are used in multiple threads at once. The TCriticalSection (348) class is a wrapper around low-level critical section routines (semaphores or mutexes). The TEventObject (350) class can be used to send messages between threads (also known as conditional variables in Posix threads).

29.3 Constants, types and variables

29.3.1 Constants

```
INFINITE = Cardinal ( - 1 )
```

Constant denoting an infinite timeout.

29.3.2 Types

```
PSecurityAttributes = Pointer
```

`PSecurityAttributes` is a dummy type used in non-windows implementations, so the calls remain Delphi compatible.

```
TEvent = TEventObject
```

`TEvent` is a simple alias for the TEventObject (350) class.

```
TEventHandle = Pointer
```

`TEventHandle` is an opaque type and should not be used in user code.

```
TWaitResult = (wrSignaled,wrTimeout,wrAbandoned,wrError)
```

Table 29.2: Enumeration values for type TWaitResult

Value	Explanation
wrAbandoned	Wait operation was abandoned.
wrError	An error occurred during the wait operation.
wrSignaled	Event was signaled (triggered)
wrTimeout	Time-out period expired

`TWaitResult` is used to report the result of a wait operation.

29.4 TCriticalSection

29.4.1 Description

`TCriticalSection` is a class wrapper around the low-level `TRTLCriticalSection` routines. It simply calls the RTL routines in the system unit for critical section support.

A critical section is a resource which can be owned by only 1 caller: it can be used to make sure that in a multithreaded application only 1 thread enters pieces of code protected by the critical section.

Typical usage is to protect a piece of code with the following code (`MySection` is a `TCriticalSection` instance):

```
// Previous code
  MySection.Acquire;
  Try
    // Protected code
  Finally
    MySection.Release;
  end;
  // Other code.
```

The protected code can be executed by only 1 thread at a time. This is useful for instance for list operations in multithreaded environments.

29.4.2 Method overview

Page	Property	Description
349	Acquire	Enter the critical section
350	Create	Create a new critical section.
350	Destroy	Destroy the criticalsection instance
349	Enter	Alias for `Acquire`
349	Leave	Alias for `Release`
349	Release	Leave the critical section

29.4.3 TCriticalSection.Acquire

Synopsis: Enter the critical section

Declaration: `procedure Acquire; Override`

Visibility: public

Description: `Acquire` attempts to enter the critical section. It will suspend the calling thread if the critical section is in use by another thread, and will resume as soon as the other thread has released the critical section.

See also: TCriticalSection.Release (349)

29.4.4 TCriticalSection.Release

Synopsis: Leave the critical section

Declaration: `procedure Release; Override`

Visibility: public

Description: `Release` leaves the critical section. It will free the critical section so another thread waiting to enter the critical section will be awakened, and will enter the critical section. This call always returns immediatly.

See also: TCriticalSection.Acquire (349)

29.4.5 TCriticalSection.Enter

Synopsis: Alias for `Acquire`

Declaration: `procedure Enter`

Visibility: public

Description: `Enter` just calls Acquire (349).

See also: TCriticalSection.Leave (349), TCriticalSection.Acquire (349)

29.4.6 TCriticalSection.Leave

Synopsis: Alias for `Release`

Declaration: `procedure Leave`

Visibility: public

Description: `Leave` just calls Release (349)

See also: TCriticalSection.Release (349), TCriticalSection.Enter (349)

29.4.7 TCriticalSection.Create

Synopsis: Create a new critical section.

Declaration: `constructor Create`

Visibility: public

Description: `Create` initializes a new critical section, and initializes the system objects for the critical section. It should be created only once for all threads, all threads should use the same critical section instance.

See also: TCriticalSection.Destroy (350)

29.4.8 TCriticalSection.Destroy

Synopsis: Destroy the criticalsection instance

Declaration: `destructor Destroy;  Override`

Visibility: public

Description: `Destroy` releases the system critical section resources, and removes the `TCriticalSection` instance from memory.

Errors: Any threads trying to enter the critical section when it is destroyed, will start running with an error (an exception should be raised).

See also: TCriticalSection.Create (350), TCriticalSection.Acquire (349)

29.5 TEventObject

29.5.1 Description

`TEventObject` encapsulates the `BasicEvent` implementation of the system unit in a class. The event can be used to notify other threads of a change in conditions. (in POSIX terms, this is a conditional variable). A thread that wishes to notify other threads creates an instance of `TEventObject` with a certain name, and posts events to it. Other threads that wish to be notified of these events should create their own instances of `TEventObject` with the same name, and wait for events to arrive.

29.5.2 Method overview

Page	Property	Description
351	Create	Create a new event object
351	destroy	Clean up the event and release from memory
351	ResetEvent	Reset the event
351	SetEvent	Set the event
352	WaitFor	Wait for the event to be set.

29.5.3 Property overview

Page	Property	Access	Description
352	ManualReset	r	Should the event be reset manually

29.5.4 TEventObject.Create

Synopsis: Create a new event object

Declaration:
```
constructor Create(EventAttributes: PSecurityAttributes;
                   AManualReset: Boolean;InitialState: Boolean;
                   const Name: String)
```

Visibility: public

Description: `Create` creates a new event object with unique name `AName`. The object will be created security attributes `EventAttributes` (windows only).

The `AManualReset` indicates whether the event must be reset manually (if it is `False`, the event is reset immediatly after the first thread waiting for it is notified). `InitialState` determines whether the event is initially set or not.

See also: TEventObject.ManualReset (352), TEventObject.ResetEvent (351)

29.5.5 TEventObject.destroy

Synopsis: Clean up the event and release from memory

Declaration:
```
destructor destroy;  Override
```

Visibility: public

Description: `Destroy` cleans up the low-level resources allocated for this event and releases the event instance from memory.

See also: TEventObject.Create (351)

29.5.6 TEventObject.ResetEvent

Synopsis: Reset the event

Declaration:
```
procedure ResetEvent
```

Visibility: public

Description: `ResetEvent` turns off the event. Any WaitFor (352) operation will suspend the calling thread.

See also: TEventObject.SetEvent (351), TEventObject.WaitFor (352)

29.5.7 TEventObject.SetEvent

Synopsis: Set the event

Declaration:
```
procedure SetEvent
```

Visibility: public

Description: `SetEvent` sets the event. If the ManualReset (352) is `True` any thread that was waiting for the event to be set (using WaitFor (352)) will resume it's operation. After the event was set, any thread that executes `WaitFor` will return at once. If `ManualReset` is `False`, only one thread will be notified that the event was set, and the event will be immediatly reset after that.

See also: TEventObject.WaitFor (352), TEventObject.ManualReset (352)

29.5.8 TEventObject.WaitFor

Synopsis: Wait for the event to be set.

Declaration: `function WaitFor(Timeout: Cardinal) : TWaitResult`

Visibility: public

Description: `WaitFor` should be used in threads that should be notified when the event is set. When `WaitFor` is called, and the event is not set, the thread will be suspended. As soon as the event is set by some other thread (using SetEvent (351)) or the timeout period (`TimeOut`) has expired, the `WaitFor` function returns. The return value depends on the condition that caused the `WaitFor` function to return.

The calling thread will wait indefinitely when the constant `INFINITE` is specified for the `TimeOut` parameter.

See also: TEventObject.SetEvent (351)

29.5.9 TEventObject.ManualReset

Synopsis: Should the event be reset manually

Declaration: `Property ManualReset : Boolean`

Visibility: public

Access: Read

Description: Should the event be reset manually

29.6 THandleObject

29.6.1 Description

`THandleObject` is a parent class for synchronization classes that need to store an operating system handle. It introduces a property Handle (353) which can be used to store the operating system handle. The handle is in no way manipulated by `THandleObject`, only storage is provided.

29.6.2 Method overview

Page	Property	Description
352	destroy	Free the instance

29.6.3 Property overview

Page	Property	Access	Description
353	Handle	r	Handle for this object
353	LastError	r	Last operating system error

29.6.4 THandleObject.destroy

Synopsis: Free the instance

Declaration: `destructor destroy;  Override`

Visibility: public

Description: `Destroy` does nothing in the Free Pascal implementation of `THandleObject`.

29.6.5 THandleObject.Handle

Synopsis: Handle for this object

Declaration: `Property Handle : TEventHandle`

Visibility: public

Access: Read

Description: `Handle` provides read-only access to the operating system handle of this instance. The public access is read-only, descendent classes should set the handle by accessing it's protected field `FHandle` directly.

29.6.6 THandleObject.LastError

Synopsis: Last operating system error

Declaration: `Property LastError : Integer`

Visibility: public

Access: Read

Description: `LastError` provides read-only access to the last operating system error code for operations on Handle (353).

See also: THandleObject.Handle (353)

29.7 TSimpleEvent

29.7.1 Description

`TSimpleEvent` is a simple descendent of the TEventObject (350) class. It creates an event with no name, which must be reset manually, and which is initially not set.

29.7.2 Method overview

Page	Property	Description
353	Create	Creates a new `TSimpleEvent` instance

29.7.3 TSimpleEvent.Create

Synopsis: Creates a new `TSimpleEvent` instance

Declaration: `constructor Create`

Visibility: default

Description: `Create` instantiates a new `TSimpleEvent` instance. It simply calls the inherited Create (351) with `Nil` for the security attributes, an empty name, `AManualReset` set to `True`, and `InitialState` to `False`.

See also: TEventObject.Create (351)

29.8 TSynchroObject

29.8.1 Description

`TSynchroObject` is an abstract synchronization resource object. It implements 2 virtual methods Acquire (354) which can be used to acquire the resource, and Release (354) to release the resource.

29.8.2 Method overview

Page	Property	Description
354	Acquire	Acquire synchronization resource
354	Release	Release previously acquired synchronization resource

29.8.3 TSynchroObject.Acquire

Synopsis: Acquire synchronization resource

Declaration: `procedure Acquire; Virtual`

Visibility: default

Description: `Acquire` does nothing in `TSynchroObject`. Descendent classes must override this method to acquire the resource they manage.

See also: TSynchroObject.Release (354)

29.8.4 TSynchroObject.Release

Synopsis: Release previously acquired synchronization resource

Declaration: `procedure Release; Virtual`

Visibility: default

Description: `Release` does nothing in `TSynchroObject`. Descendent classes must override this method to release the resource they acquired through the Acquire (354) call.

See also: TSynchroObject.Acquire (354)

Chapter 30

Reference for unit 'URIParser'

30.1 Overview

The URIParser unit contains a basic type (TURI (355)) and some routines for the parsing (ParseURI (356)) and construction (EncodeURI (355)) of Uniform Resource Indicators, commonly referred to as URL: Uniform Resource Location. It is used in various other units, and in itself contains no classes. It supports all protocols, username/password/port specification, query parameters and bookmarks etc..

30.2 Constants, types and variables

30.2.1 Types

```
TURI = record
  Protocol : String;
  Username : String;
  Password : String;
  Host : String;
  Port : Word;
  Path : String;
  Document : String;
  Params : String;
  Bookmark : String;
  HasAuthority : Boolean;
end
```

`TURI` is the basic record that can be filled by the ParseURI (356) call. It contains the contents of a URI, parsed out in it's various pieces.

30.3 Procedures and functions

30.3.1 EncodeURI

Synopsis: Form a string representation of the URI

Declaration: `function EncodeURI(const URI: TURI) : String`

Visibility: default

Description: `EncodeURI` will return a valid text representation of the URI in the `URI` record.

See also: ParseURI (356)

30.3.2 FilenameToURI

Synopsis: Construct a URI from a filename

Declaration:
```
function FilenameToURI(const Filename: String) : String
```

Visibility: default

Description: `FilenameToURI` takes `Filename` and constructs a `file:` protocol URI from it.

Errors: None.

See also: URIToFilename (357)

30.3.3 IsAbsoluteURI

Synopsis: Check whether a URI is absolute.

Declaration:
```
function IsAbsoluteURI(const UriReference: String) : Boolean
```

Visibility: default

Description: `IsAbsoluteURI` returns `True` if the URI in `UriReference` is absolute, i.e. contains a protocol part.

Errors: None.

See also: FilenameToURI (356), URIToFileName (357)

30.3.4 ParseURI

Synopsis: Parse a URI and split it into its constituent parts

Declaration:
```
function ParseURI(const URI: String) : TURI;  Overload
function ParseURI(const URI: String;const DefaultProtocol: String;
                  DefaultPort: Word) : TURI;  Overload
```

Visibility: default

Description: `ParseURI` decodes `URI` and returns the various parts of the URI in the result record.

The function accepts the most general URI scheme:

```
proto://user:pwd@host:port/path/document?params#bookmark
```

Missing (optional) parts in the URI will be left blank in the result record. If a default protocol and port are specified, they will be used in the record if the corresponding part is not present in the URI.

See also: EncodeURI (355)

30.3.5 ResolveRelativeURI

Synopsis: Return a relative link

Declaration:
```
function ResolveRelativeURI(const BaseUri: WideString;
                            const RelUri: WideString;
                            out ResultUri: WideString) : Boolean
                            ;  Overload
function ResolveRelativeURI(const BaseUri: UTF8String;
                            const RelUri: UTF8String;
                            out ResultUri: UTF8String) : Boolean
                            ;  Overload
```

Visibility: default

Description: `ResolveRelativeURI` returns in `ResultUri` an absolute link constructed from a base URI `BaseURI` and a relative link `RelURI`. One of the two URI names must have a protocol specified. If the `RelURI` argument contains a protocol, it is considered a complete (absolute) URI and is returned as the result.

The function returns `True` if a link was succesfully returned.

Errors: If no protocols are specified, the function returns `False`

30.3.6 URIToFilename

Synopsis: Convert a URI to a filename

Declaration:
```
function URIToFilename(const URI: String;out Filename: String) : Boolean
```

Visibility: default

Description: `URIToFilename` returns a filename (using the correct Path Delimiter character) from `URI`. The `URI` must be of protocol `File` or have no protocol.

Errors: If the URI contains an unsupported protocol, `False` is returned.

See also: ResolveRelativeURI (357), FilenameToURI (356)

Chapter 31

Reference for unit 'zstream'

31.1 Used units

Table 31.1: Used units by unit 'zstream'

Name	Page
Classes	**??**
gzio	358
zbase	358

31.2 Overview

The ZStream unit implements a TStream (**??**) descendent (TCompressionStream (359)) which uses the deflate algorithm to compress everything that is written to it. The compressed data is written to the output stream, which is specified when the compressor class is created.

Likewise, a `TStream` descendent is implemented which reads data from an input stream (TDecompressionStream (362)) and decompresses it with the inflate algorithm.

31.3 Constants, types and variables

31.3.1 Types

```
Tcompressionlevel = (clnone,clfastest,cldefault,clmax)
```

Compression level for the deflate algorithm

```
Tgzopenmode = (gzopenread,gzopenwrite)
```

Open mode for gzip file.

Table 31.2: Enumeration values for type Tcompressionlevel

Value	Explanation
cldefault	Use default compression
clfastest	Use fast (but less) compression.
clmax	Use maximum compression
clnone	Do not use compression, just copy data.

Table 31.3: Enumeration values for type Tgzopenmode

Value	Explanation
gzopenread	Open file for reading
gzopenwrite	Open file for writing

31.4 Ecompressionerror

31.4.1 Description

`ECompressionError` is the exception class used by the TCompressionStream (359) class.

31.5 Edecompressionerror

31.5.1 Description

`EDecompressionError` is the exception class used by the TDeCompressionStream (362) class.

31.6 Egzfileerror

31.6.1 Description

`Egzfileerror` is the exception class used to report errors by the Tgzfilestream (364) class.

31.7 Ezliberror

31.7.1 Description

Errors which occur in the zstream unit are signaled by raising an `EZLibError` exception descendent.

31.8 Tcompressionstream

31.8.1 Description

`TCompressionStream`

31.8.2 Method overview

Page	Property	Description
360	create	Create a new instance of the compression stream.
360	destroy	Flushe data to the output stream and destroys the compression stream.
361	flush	Flush remaining data to the target stream
361	get_compressionrate	Get the current compression rate
360	write	Write data to the stream

31.8.3 Tcompressionstream.create

Synopsis: Create a new instance of the compression stream.

Declaration:
```
constructor create(level: Tcompressionlevel;dest: TStream;
                   Askipheader: Boolean)
```

Visibility: public

Description: `Create` creates a new instance of the compression stream. It merely calls the inherited constructor with the destination stream `Dest` and stores the compression level.

If `ASkipHeader` is set to `True`, the method will not write the block header to the stream. This is required for deflated data in a zip file.

Note that the compressed data is only completely written after the compression stream is destroyed.

See also: TCompressionStream.Destroy (360)

31.8.4 Tcompressionstream.destroy

Synopsis: Flushe data to the output stream and destroys the compression stream.

Declaration:
```
destructor destroy;  Override
```

Visibility: public

Description: `Destroy` flushes the output stream: any compressed data not yet written to the output stream are written, and the deflate structures are cleaned up.

Errors: None.

See also: TCompressionStream.Create (360)

31.8.5 Tcompressionstream.write

Synopsis: Write data to the stream

Declaration:
```
function write(const buffer;count: LongInt) : LongInt;  Override
```

Visibility: public

Description: `Write` takes `Count` bytes from `Buffer` and comresseses (deflates) them. The compressed result is written to the output stream.

Errors: If an error occurs, an ECompressionError (359) exception is raised.

See also: TCompressionStream.Read (359), TCompressionStream.Seek (359)

31.8.6 Tcompressionstream.flush

Synopsis: Flush remaining data to the target stream

Declaration: `procedure flush`

Visibility: public

Description: `flush` writes any remaining data in the memory buffers to the target stream, and clears the memory buffer.

31.8.7 Tcompressionstream.get_compressionrate

Synopsis: Get the current compression rate

Declaration: `function get_compressionrate : single`

Visibility: public

Description: `get_compressionrate` returns the percentage of the number of written compressed bytes relative to the number of written bytes.

Errors: If no bytes were written, an exception is raised.

31.9 Tcustomzlibstream

31.9.1 Description

`TCustomZlibStream` serves as the ancestor class for the TCompressionStream (359) and TDecompressionStream (362) classes.

It introduces support for a progess handler, and stores the input or output stream.

31.9.2 Method overview

Page	Property	Description
361	create	Create a new instance of `TCustomZlibStream`
362	destroy	Clear up instance

31.9.3 Tcustomzlibstream.create

Synopsis: Create a new instance of `TCustomZlibStream`

Declaration: `constructor create(stream: TStream)`

Visibility: public

Description: `Create` creates a new instance of `TCustomZlibStream`. It stores a reference to the input/output stream, and initializes the deflate compression mechanism so they can be used by the descendents.

See also: TCompressionStream (359), TDecompressionStream (362)

31.9.4 Tcustomzlibstream.destroy

Synopsis: Clear up instance

Declaration: `destructor destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the internal memory buffer and calls the inherited destroy.

See also: Tcustomzlibstream.create (361)

31.10 Tdecompressionstream

31.10.1 Description

`TDecompressionStream` performs the inverse operation of TCompressionStream (359). A read operation reads data from an input stream and decompresses (inflates) the data it as it goes along.

The decompression stream reads it's compressed data from a stream with deflated data. This data can be created e.g. with a TCompressionStream (359) compression stream.

31.10.2 Method overview

Page	Property	Description
362	create	Creates a new instance of the `TDecompressionStream` stream
362	destroy	Destroys the `TDecompressionStream` instance
364	get_compressionrate	Get the current compression rate
363	read	Read data from the compressed stream
363	seek	Move stream position to a certain location in the stream.

31.10.3 Tdecompressionstream.create

Synopsis: Creates a new instance of the `TDecompressionStream` stream

Declaration: `constructor create(Asource: TStream;Askipheader: Boolean)`

Visibility: public

Description: `Create` creates and initializes a new instance of the `TDecompressionStream` class. It calls the inherited `Create` and passes it the `Source` stream. The source stream is the stream from which the compressed (deflated) data is read.

If `ASkipHeader` is true, then the gzip data header is skipped, allowing `TDecompressionStream` to read deflated data in a .zip file. (this data does not have the gzip header record prepended to it).

Note that the source stream is by default not owned by the decompression stream, and is not freed when the decompression stream is destroyed.

See also: TDecompressionStream.Destroy (362)

31.10.4 Tdecompressionstream.destroy

Synopsis: Destroys the `TDecompressionStream` instance

Declaration: `destructor destroy;  Override`

Visibility: public

Description: `Destroy` cleans up the inflate structure, and then simply calls the inherited destroy.

By default the source stream is not freed when calling `Destroy`.

See also: TDecompressionStream.Create (362)

31.10.5 Tdecompressionstream.read

Synopsis: Read data from the compressed stream

Declaration: `function read(var buffer;count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` will read data from the compressed stream until the decompressed data size is `Count` or there is no more compressed data available. The decompressed data is written in `Buffer`. The function returns the number of bytes written in the buffer.

Errors: If an error occurs, an EDeCompressionError (359) exception is raised.

See also: TCompressionStream.Write (360)

31.10.6 Tdecompressionstream.seek

Synopsis: Move stream position to a certain location in the stream.

Declaration: `function seek(offset: LongInt;origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` overrides the standard `Seek` implementation. There are a few differences between the implementation of `Seek` in Free Pascal compared to Delphi:

- In Free Pascal, you can perform any seek. In case of a forward seek, the Free Pascal implementation will read some bytes until the desired position is reached, in case of a backward seek it will seek the source stream backwards to the position it had at the creation time of the `TDecompressionStream` and then again read some bytes until the desired position has been reached.
- In Free Pascal, a seek with soFromBeginning will reset the source stream to the position it had when the `TDecompressionStream` was created. In Delphi, the source stream is reset to position 0. This means that at creation time the source stream must always be at the start of the zstream, you cannot use `TDecompressionStream.Seek` to reset the source stream to the begin of the file.

Errors: An EDecompressionError (359) exception is raised if the stream does not allow the requested seek operation.

See also: TDecompressionStream.Read (363)

31.10.7 Tdecompressionstream.get_compressionrate

Synopsis: Get the current compression rate

Declaration: `function get_compressionrate : single`

Visibility: public

Description: `get_compressionrate` returns the percentage of the number of read compressed bytes relative to the total number of read bytes.

Errors: If no bytes were written, an exception is raised.

31.11 TGZFileStream

31.11.1 Description

`TGZFileStream` can be used to read data from a gzip file, or to write data to a gzip file.

31.11.2 Method overview

Page	Property	Description
364	create	Create a new instance of `TGZFileStream`
365	destroy	Removes `TGZFileStream` instance
364	read	Read data from the compressed file
365	seek	Set the position in the compressed stream.
365	write	Write data to be compressed

31.11.3 TGZFileStream.create

Synopsis: Create a new instance of `TGZFileStream`

Declaration: `constructor create(filename: ansistring;filemode: Tgzopenmode)`

Visibility: public

Description: `Create` creates a new instance of the `TGZFileStream` class. It opens `FileName` for reading or writing, depending on the `FileMode` parameter. It is not possible to open the file read-write. If the file is opened for reading, it must exist.

If the file is opened for reading, the TGZFileStream.Read (364) method can be used for reading the data in uncompressed form.

If the file is opened for writing, any data written using the TGZFileStream.Write (365) method will be stored in the file in compressed (deflated) form.

Errors: If the file is not found, an EZlibError (359) exception is raised.

See also: TGZFileStream.Destroy (365), TGZOpenMode (358)

31.11.4 TGZFileStream.read

Synopsis: Read data from the compressed file

Declaration: `function read(var buffer;count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Read` overrides the `Read` method of `TStream` to read the data from the compressed file. The `Buffer` parameter indicates where the read data should be stored. The `Count` parameter specifies the number of bytes (*uncompressed*) that should be read from the compressed file. Note that it is not possible to read from the stream if it was opened in write mode.

The function returns the number of uncompressed bytes actually read.

Errors: If `Buffer` points to an invalid location, or does not have enough room for `Count` bytes, an exception will be raised.

See also: TGZFileStream.Create (364), TGZFileStream.Write (365), TGZFileStream.Seek (365)

31.11.5 TGZFileStream.write

Synopsis: Write data to be compressed

Declaration: `function write(const buffer;count: LongInt) : LongInt;  Override`

Visibility: public

Description: `Write` writes `Count` bytes from `Buffer` to the compressed file. The data is compressed as it is written, so ideally, less than `Count` bytes end up in the compressed file. Note that it is not possible to write to the stream if it was opened in read mode.

The function returns the number of (uncompressed) bytes that were actually written.

Errors: In case of an error, an EZlibError (359) exception is raised.

See also: TGZFileStream.Create (364), TGZFileStream.Read (364), TGZFileStream.Seek (365)

31.11.6 TGZFileStream.seek

Synopsis: Set the position in the compressed stream.

Declaration: `function seek(offset: LongInt;origin: Word) : LongInt;  Override`

Visibility: public

Description: `Seek` sets the position to `Offset` bytes, starting from `Origin`. Not all combinations are possible, see TDecompressionStream.Seek (363) for a list of possibilities.

Errors: In case an impossible combination is asked, an EZlibError (359) exception is raised.

See also: TDecompressionStream.Seek (363)

31.11.7 TGZFileStream.destroy

Synopsis: Removes `TGZFileStream` instance

Declaration: `destructor destroy;  Override`

Visibility: public

Description: `Destroy` closes the file and releases the `TGZFileStream` instance from memory.

See also: TGZFileStream.Create (364)