

Free Component Library (FCL):  
Reference guide.

---

Reference guide for FCL units.  
Document version 2.1  
October 2009

Michaël Van Canneyt

---

# Contents

0.1	Overview	30
<b>1</b>	<b>Reference for unit 'AVL_Tree'</b>	<b>31</b>
1.1	Used units	31
1.2	Overview	31
1.3	TAVLTree	31
1.3.1	Description	31
1.3.2	Method overview	32
1.3.3	Property overview	32
1.3.4	TAVLTree.Find	32
1.3.5	TAVLTree.FindKey	33
1.3.6	TAVLTree.FindSuccessor	33
1.3.7	TAVLTree.FindPrecessor	33
1.3.8	TAVLTree.FindLowest	33
1.3.9	TAVLTree.FindHighest	34
1.3.10	TAVLTree.FindNearest	34
1.3.11	TAVLTree.FindPointer	34
1.3.12	TAVLTree.FindLeftMost	34
1.3.13	TAVLTree.FindRightMost	35
1.3.14	TAVLTree.FindLeftMostKey	35
1.3.15	TAVLTree.FindRightMostKey	35
1.3.16	TAVLTree.FindLeftMostSameKey	35
1.3.17	TAVLTree.FindRightMostSameKey	36
1.3.18	TAVLTree.Add	36
1.3.19	TAVLTree.Delete	36
1.3.20	TAVLTree.Remove	36
1.3.21	TAVLTree.RemovePointer	37
1.3.22	TAVLTree.MoveDataLeftMost	37
1.3.23	TAVLTree.MoveDataRightMost	37
1.3.24	TAVLTree.Clear	37
1.3.25	TAVLTree.FreeAndClear	38

1.3.26	TAVLTree.FreeAndDelete	38
1.3.27	TAVLTree.ConsistencyCheck	38
1.3.28	TAVLTree.WriteReportToStream	38
1.3.29	TAVLTree.ReportAsString	39
1.3.30	TAVLTree.Create	39
1.3.31	TAVLTree.Destroy	39
1.3.32	TAVLTree.OnCompare	39
1.3.33	TAVLTree.Count	40
1.4	TAVLTreeNode	40
1.4.1	Description	40
1.4.2	Method overview	40
1.4.3	TAVLTreeNode.Clear	40
1.4.4	TAVLTreeNode.TreeDepth	40
1.5	TAVLTreeNodeMemManager	40
1.5.1	Description	40
1.5.2	Method overview	41
1.5.3	Property overview	41
1.5.4	TAVLTreeNodeMemManager.DisposeNode	41
1.5.5	TAVLTreeNodeMemManager.NewNode	41
1.5.6	TAVLTreeNodeMemManager.Clear	41
1.5.7	TAVLTreeNodeMemManager.Create	42
1.5.8	TAVLTreeNodeMemManager.Destroy	42
1.5.9	TAVLTreeNodeMemManager.MinimumFreeNode	42
1.5.10	TAVLTreeNodeMemManager.MaximumFreeNodeRatio	42
1.5.11	TAVLTreeNodeMemManager.Count	43
<b>2</b>	<b>Reference for unit 'base64'</b>	<b>44</b>
2.1	Used units	44
2.2	Overview	44
2.3	Constants, types and variables	44
2.3.1	Types	44
2.4	EBase64DecodingException	45
2.4.1	Description	45
2.5	TBase64DecodingStream	45
2.5.1	Description	45
2.5.2	Method overview	45
2.5.3	Property overview	45
2.5.4	TBase64DecodingStream.Create	45
2.5.5	TBase64DecodingStream.Reset	46
2.5.6	TBase64DecodingStream.Read	46

2.5.7	TBase64DecodingStream.Seek . . . . .	46
2.5.8	TBase64DecodingStream.EOF . . . . .	47
2.5.9	TBase64DecodingStream.Mode . . . . .	47
2.6	TBase64EncodingStream . . . . .	47
2.6.1	Description . . . . .	47
2.6.2	Method overview . . . . .	47
2.6.3	TBase64EncodingStream.Create . . . . .	47
2.6.4	TBase64EncodingStream.Destroy . . . . .	48
2.6.5	TBase64EncodingStream.Write . . . . .	48
2.6.6	TBase64EncodingStream.Seek . . . . .	48
<b>3</b>	<b>Reference for unit 'BlowFish' . . . . .</b>	<b>49</b>
3.1	Used units . . . . .	49
3.2	Overview . . . . .	49
3.3	Constants, types and variables . . . . .	49
3.3.1	Constants . . . . .	49
3.3.2	Types . . . . .	49
3.4	EBlowFishError . . . . .	50
3.4.1	Description . . . . .	50
3.5	TBlowFish . . . . .	50
3.5.1	Description . . . . .	50
3.5.2	Method overview . . . . .	50
3.5.3	TBlowFish.Create . . . . .	50
3.5.4	TBlowFish.Encrypt . . . . .	50
3.5.5	TBlowFish.Decrypt . . . . .	51
3.6	TBlowFishDeCryptStream . . . . .	51
3.6.1	Description . . . . .	51
3.6.2	Method overview . . . . .	51
3.6.3	TBlowFishDeCryptStream.Read . . . . .	51
3.6.4	TBlowFishDeCryptStream.Seek . . . . .	52
3.7	TBlowFishEncryptStream . . . . .	52
3.7.1	Description . . . . .	52
3.7.2	Method overview . . . . .	52
3.7.3	TBlowFishEncryptStream.Destroy . . . . .	52
3.7.4	TBlowFishEncryptStream.Write . . . . .	52
3.7.5	TBlowFishEncryptStream.Seek . . . . .	53
3.7.6	TBlowFishEncryptStream.Flush . . . . .	53
3.8	TBlowFishStream . . . . .	53
3.8.1	Description . . . . .	53
3.8.2	Method overview . . . . .	54

3.8.3	Property overview	54
3.8.4	TBlowFishStream.Create	54
3.8.5	TBlowFishStream.Destroy	54
3.8.6	TBlowFishStream.BlowFish	54
<b>4</b>	<b>Reference for unit 'bufstream'</b>	<b>55</b>
4.1	Used units	55
4.2	Overview	55
4.3	Constants, types and variables	55
4.3.1	Constants	55
4.4	TBufStream	55
4.4.1	Description	55
4.4.2	Method overview	56
4.4.3	Property overview	56
4.4.4	TBufStream.Create	56
4.4.5	TBufStream.Destroy	56
4.4.6	TBufStream.Buffer	56
4.4.7	TBufStream.Capacity	57
4.4.8	TBufStream.BufferPos	57
4.4.9	TBufStream.BufferSize	57
4.5	TReadBufStream	58
4.5.1	Description	58
4.5.2	Method overview	58
4.5.3	TReadBufStream.Seek	58
4.5.4	TReadBufStream.Read	58
4.6	TWriteBufStream	58
4.6.1	Description	58
4.6.2	Method overview	59
4.6.3	TWriteBufStream.Destroy	59
4.6.4	TWriteBufStream.Seek	59
4.6.5	TWriteBufStream.Write	59
<b>5</b>	<b>Reference for unit 'CacheCls'</b>	<b>60</b>
5.1	Used units	60
5.2	Overview	60
5.3	Constants, types and variables	60
5.3.1	Resource strings	60
5.3.2	Types	60
5.4	ECacheError	61
5.4.1	Description	61
5.5	TCache	61

5.5.1	Description	61
5.5.2	Method overview	62
5.5.3	Property overview	62
5.5.4	TCache.Create	62
5.5.5	TCache.Destroy	62
5.5.6	TCache.Add	62
5.5.7	TCache.AddNew	63
5.5.8	TCache.FindSlot	63
5.5.9	TCache.IndexOf	63
5.5.10	TCache.Remove	64
5.5.11	TCache.Data	64
5.5.12	TCache.MRUSlot	64
5.5.13	TCache.LRUSlot	65
5.5.14	TCache.SlotCount	65
5.5.15	TCache.Slots	65
5.5.16	TCache.OnIsDataEqual	65
5.5.17	TCache.OnFreeSlot	66
<b>6</b>	<b>Reference for unit 'contnrs'</b>	<b>67</b>
6.1	Used units	67
6.2	Overview	67
6.3	Constants, types and variables	67
6.3.1	Constants	67
6.3.2	Types	68
6.4	Procedures and functions	70
6.4.1	RSHash	70
6.5	EDuplicate	71
6.5.1	Description	71
6.6	EKeyNotFound	71
6.6.1	Description	71
6.7	TBucketList	71
6.7.1	Method overview	71
6.7.2	TBucketList.Create	71
6.8	TClassList	71
6.8.1	Description	71
6.8.2	Method overview	72
6.8.3	Property overview	72
6.8.4	TClassList.Add	72
6.8.5	TClassList.Extract	72
6.8.6	TClassList.Remove	72

6.8.7	TClassList.IndexOf	73
6.8.8	TClassList.First	73
6.8.9	TClassList.Last	73
6.8.10	TClassList.Insert	74
6.8.11	TClassList.Items	74
6.9	TComponentList	74
6.9.1	Description	74
6.9.2	Method overview	74
6.9.3	Property overview	74
6.9.4	TComponentList.Destroy	75
6.9.5	TComponentList.Add	75
6.9.6	TComponentList.Extract	75
6.9.7	TComponentList.Remove	75
6.9.8	TComponentList.IndexOf	76
6.9.9	TComponentList.First	76
6.9.10	TComponentList.Last	76
6.9.11	TComponentList.Insert	77
6.9.12	TComponentList.Items	77
6.10	TCustomBucketList	77
6.10.1	Method overview	77
6.10.2	Property overview	77
6.10.3	TCustomBucketList.Destroy	77
6.10.4	TCustomBucketList.Clear	78
6.10.5	TCustomBucketList.Add	78
6.10.6	TCustomBucketList.Assign	78
6.10.7	TCustomBucketList.Exists	78
6.10.8	TCustomBucketList.Find	78
6.10.9	TCustomBucketList.ForEach	78
6.10.10	TCustomBucketList.Remove	78
6.10.11	TCustomBucketList.Data	78
6.11	TFPCustomHashTable	79
6.11.1	Description	79
6.11.2	Method overview	79
6.11.3	Property overview	79
6.11.4	TFPCustomHashTable.Create	79
6.11.5	TFPCustomHashTable.CreateWith	80
6.11.6	TFPCustomHashTable.Destroy	80
6.11.7	TFPCustomHashTable.ChangeTableSize	80
6.11.8	TFPCustomHashTable.Clear	80
6.11.9	TFPCustomHashTable.Delete	81

6.11.10 TFPCustomHashTable.Find	81
6.11.11 TFPCustomHashTable.IsEmpty	81
6.11.12 TFPCustomHashTable.HashFunction	81
6.11.13 TFPCustomHashTable.Count	82
6.11.14 TFPCustomHashTable.HashTableSize	82
6.11.15 TFPCustomHashTable.HashTable	82
6.11.16 TFPCustomHashTable.VoidSlots	83
6.11.17 TFPCustomHashTable.LoadFactor	83
6.11.18 TFPCustomHashTable.AVGChainLen	83
6.11.19 TFPCustomHashTable.MaxChainLength	83
6.11.20 TFPCustomHashTable.NumberOfCollisions	84
6.11.21 TFPCustomHashTable.Density	84
6.12 TFPDDataHashTable	84
6.12.1 Description	84
6.12.2 Method overview	84
6.12.3 Property overview	85
6.12.4 TFPDDataHashTable.Add	85
6.12.5 TFPDDataHashTable.Items	85
6.13 TFPHashList	85
6.13.1 Description	85
6.13.2 Method overview	86
6.13.3 Property overview	86
6.13.4 TFPHashList.Create	86
6.13.5 TFPHashList.Destroy	86
6.13.6 TFPHashList.Add	87
6.13.7 TFPHashList.Clear	87
6.13.8 TFPHashList.NameOfIndex	87
6.13.9 TFPHashList.HashOfIndex	87
6.13.10 TFPHashList.Delete	88
6.13.11 TFPHashList.Error	88
6.13.12 TFPHashList.Expand	88
6.13.13 TFPHashList.Extract	88
6.13.14 TFPHashList.IndexOf	89
6.13.15 TFPHashList.Find	89
6.13.16 TFPHashList.FindIndexOf	89
6.13.17 TFPHashList.FindWithHash	89
6.13.18 TFPHashList.Rename	90
6.13.19 TFPHashList.Remove	90
6.13.20 TFPHashList.Pack	90
6.13.21 TFPHashList.ShowStatistics	90

6.13.22	TFPHashList.ForEachCall	91
6.13.23	TFPHashList.Capacity	91
6.13.24	TFPHashList.Count	91
6.13.25	TFPHashList.Items	91
6.13.26	TFPHashList.List	92
6.13.27	TFPHashList.Strs	92
6.14	TFPHashObject	92
6.14.1	Description	92
6.14.2	Method overview	92
6.14.3	Property overview	92
6.14.4	TFPHashObject.CreateNotOwned	93
6.14.5	TFPHashObject.Create	93
6.14.6	TFPHashObject.ChangeOwner	93
6.14.7	TFPHashObject.ChangeOwnerAndName	93
6.14.8	TFPHashObject.Rename	94
6.14.9	TFPHashObject.Name	94
6.14.10	TFPHashObject.Hash	94
6.15	TFPHashObjectList	95
6.15.1	Method overview	95
6.15.2	Property overview	95
6.15.3	TFPHashObjectList.Create	95
6.15.4	TFPHashObjectList.Destroy	95
6.15.5	TFPHashObjectList.Clear	96
6.15.6	TFPHashObjectList.Add	96
6.15.7	TFPHashObjectList.NameOfIndex	96
6.15.8	TFPHashObjectList.HashOfIndex	97
6.15.9	TFPHashObjectList.Delete	97
6.15.10	TFPHashObjectList.Expand	97
6.15.11	TFPHashObjectList.Extract	97
6.15.12	TFPHashObjectList.Remove	98
6.15.13	TFPHashObjectList.IndexOf	98
6.15.14	TFPHashObjectList.Find	98
6.15.15	TFPHashObjectList.FindIndexOf	98
6.15.16	TFPHashObjectList.FindWithHash	99
6.15.17	TFPHashObjectList.Rename	99
6.15.18	TFPHashObjectList.FindInstanceOf	99
6.15.19	TFPHashObjectList.Pack	99
6.15.20	TFPHashObjectList.ShowStatistics	100
6.15.21	TFPHashObjectList.ForEachCall	100
6.15.22	TFPHashObjectList.Capacity	100

6.15.23	TFPHashObjectList.Count	100
6.15.24	TFPHashObjectList.OwnsObjects	101
6.15.25	TFPHashObjectList.Items	101
6.15.26	TFPHashObjectList.List	101
6.16	TFPObjectHashTable	101
6.16.1	Description	101
6.16.2	Method overview	102
6.16.3	Property overview	102
6.16.4	TFPObjectHashTable.Create	102
6.16.5	TFPObjectHashTable.CreateWith	102
6.16.6	TFPObjectHashTable.Add	103
6.16.7	TFPObjectHashTable.Items	103
6.16.8	TFPObjectHashTable.OwnsObjects	103
6.17	TFPObjectList	103
6.17.1	Description	103
6.17.2	Method overview	104
6.17.3	Property overview	104
6.17.4	TFPObjectList.Create	104
6.17.5	TFPObjectList.Destroy	104
6.17.6	TFPObjectList.Clear	105
6.17.7	TFPObjectList.Add	105
6.17.8	TFPObjectList.Delete	105
6.17.9	TFPObjectList.Exchange	106
6.17.10	TFPObjectList.Expand	106
6.17.11	TFPObjectList.Extract	106
6.17.12	TFPObjectList.Remove	106
6.17.13	TFPObjectList.IndexOf	107
6.17.14	TFPObjectList.FindInstanceOf	107
6.17.15	TFPObjectList.Insert	107
6.17.16	TFPObjectList.First	108
6.17.17	TFPObjectList.Last	108
6.17.18	TFPObjectList.Move	108
6.17.19	TFPObjectList.Assign	108
6.17.20	TFPObjectList.Pack	109
6.17.21	TFPObjectList.Sort	109
6.17.22	TFPObjectList.ForEachCall	109
6.17.23	TFPObjectList.Capacity	110
6.17.24	TFPObjectList.Count	110
6.17.25	TFPObjectList.OwnsObjects	110
6.17.26	TFPObjectList.Items	110

6.17.27	TFPObjectList.List	111
6.18	TFPStringHashTable	111
6.18.1	Description	111
6.18.2	Method overview	111
6.18.3	Property overview	111
6.18.4	TFPStringHashTable.Add	111
6.18.5	TFPStringHashTable.Items	111
6.19	THTCustomNode	112
6.19.1	Description	112
6.19.2	Method overview	112
6.19.3	Property overview	112
6.19.4	THTCustomNode.CreateWith	112
6.19.5	THTCustomNode.HasKey	112
6.19.6	THTCustomNode.Key	113
6.20	THTDataNode	113
6.20.1	Description	113
6.20.2	Property overview	113
6.20.3	THTDataNode.Data	113
6.21	THTObjectNode	113
6.21.1	Description	113
6.21.2	Property overview	113
6.21.3	THTObjectNode.Data	114
6.22	THTOwnedObjectNode	114
6.22.1	Description	114
6.22.2	Method overview	114
6.22.3	THTOwnedObjectNode.Destroy	114
6.23	THTStringNode	114
6.23.1	Description	114
6.23.2	Property overview	114
6.23.3	THTStringNode.Data	115
6.24	TObjectBucketList	115
6.24.1	Method overview	115
6.24.2	Property overview	115
6.24.3	TObjectBucketList.Add	115
6.24.4	TObjectBucketList.Remove	115
6.24.5	TObjectBucketList.Data	115
6.25	TObjectList	115
6.25.1	Description	115
6.25.2	Method overview	116
6.25.3	Property overview	116

6.25.4	TObjectList.create	116
6.25.5	TObjectList.Add	116
6.25.6	TObjectList.Extract	117
6.25.7	TObjectList.Remove	117
6.25.8	TObjectList.IndexOf	117
6.25.9	TObjectList.FindInstanceOf	117
6.25.10	TObjectList.Insert	118
6.25.11	TObjectList.First	118
6.25.12	TObjectList.Last	118
6.25.13	TObjectList.OwnsObjects	119
6.25.14	TObjectList.Items	119
6.26	TObjectQueue	119
6.26.1	Method overview	119
6.26.2	TObjectQueue.Push	119
6.26.3	TObjectQueue.Pop	120
6.26.4	TObjectQueue.Peek	120
6.27	TObjectStack	120
6.27.1	Description	120
6.27.2	Method overview	120
6.27.3	TObjectStack.Push	120
6.27.4	TObjectStack.Pop	121
6.27.5	TObjectStack.Peek	121
6.28	TOrderedList	121
6.28.1	Description	121
6.28.2	Method overview	121
6.28.3	TOrderedList.Create	121
6.28.4	TOrderedList.Destroy	122
6.28.5	TOrderedList.Count	122
6.28.6	TOrderedList.AtLeast	122
6.28.7	TOrderedList.Push	123
6.28.8	TOrderedList.Pop	123
6.28.9	TOrderedList.Peek	123
6.29	TQueue	123
6.29.1	Description	123
6.30	TStack	124
6.30.1	Description	124
<b>7</b>	<b>Reference for unit 'CustApp'</b>	<b>125</b>
7.1	Used units	125
7.2	Overview	125

7.3	Constants, types and variables	125
7.3.1	Types	125
7.4	TCustomApplication	126
7.4.1	Description	126
7.4.2	Method overview	126
7.4.3	Property overview	126
7.4.4	TCustomApplication.Create	126
7.4.5	TCustomApplication.Destroy	127
7.4.6	TCustomApplication.HandleException	127
7.4.7	TCustomApplication.Initialize	127
7.4.8	TCustomApplication.Run	128
7.4.9	TCustomApplication.ShowException	128
7.4.10	TCustomApplication.Terminate	128
7.4.11	TCustomApplication.FindOptionIndex	128
7.4.12	TCustomApplication.GetOptionValue	129
7.4.13	TCustomApplication.HasOption	129
7.4.14	TCustomApplication.CheckOptions	130
7.4.15	TCustomApplication.GetEnvironmentList	131
7.4.16	TCustomApplication.ExeName	131
7.4.17	TCustomApplication.HelpFile	131
7.4.18	TCustomApplication.Terminated	131
7.4.19	TCustomApplication.Title	132
7.4.20	TCustomApplication.OnException	132
7.4.21	TCustomApplication.ConsoleApplication	132
7.4.22	TCustomApplication.Location	132
7.4.23	TCustomApplication.Params	133
7.4.24	TCustomApplication.ParamCount	133
7.4.25	TCustomApplication.EnvironmentVariable	133
7.4.26	TCustomApplication.OptionChar	134
7.4.27	TCustomApplication.CaseSensitiveOptions	134
7.4.28	TCustomApplication.StopOnException	134
<b>8</b>	<b>Reference for unit 'dbugintf'</b>	<b>135</b>
8.1	Writing a debug server	135
8.2	Overview	135
8.3	Constants, types and variables	135
8.3.1	Resource strings	135
8.3.2	Constants	136
8.3.3	Types	136
8.4	Procedures and functions	136

8.4.1	GetDebuggingEnabled	136
8.4.2	InitDebugClient	136
8.4.3	SendBoolean	137
8.4.4	SendDateTime	137
8.4.5	SendDebug	137
8.4.6	SendDebugEx	138
8.4.7	SendDebugFmt	138
8.4.8	SendDebugFmtEx	138
8.4.9	SendInteger	138
8.4.10	SendMethodEnter	139
8.4.11	SendMethodExit	139
8.4.12	SendPointer	139
8.4.13	SendSeparator	140
8.4.14	SetDebuggingEnabled	140
8.4.15	StartDebugServer	140
<b>9</b>	<b>Reference for unit 'dbugmsg'</b>	<b>141</b>
9.1	Used units	141
9.2	Overview	141
9.3	Constants, types and variables	141
9.3.1	Constants	141
9.3.2	Types	142
9.4	Procedures and functions	142
9.4.1	DebugMessageName	142
9.4.2	ReadDebugMessageFromStream	142
9.4.3	WriteDebugMessageToStream	143
<b>10</b>	<b>Reference for unit 'eventlog'</b>	<b>144</b>
10.1	Used units	144
10.2	Overview	144
10.3	Constants, types and variables	144
10.3.1	Resource strings	144
10.3.2	Types	145
10.4	ELogError	146
10.4.1	Description	146
10.5	TEventLog	146
10.5.1	Description	146
10.5.2	Method overview	146
10.5.3	Property overview	146
10.5.4	TEventLog.Destroy	146
10.5.5	TEventLog.EventTypeToString	147

10.5.6	TEventLog.RegisterMessageFile	147
10.5.7	TEventLog.Log	148
10.5.8	TEventLog.Warning	148
10.5.9	TEventLog.Error	148
10.5.10	TEventLog.Debug	149
10.5.11	TEventLog.Info	149
10.5.12	TEventLog.Identification	149
10.5.13	TEventLog.LogType	149
10.5.14	TEventLog.Active	150
10.5.15	TEventLog.RaiseExceptionOnError	150
10.5.16	TEventLog.DefaultEventType	150
10.5.17	TEventLog.FileName	150
10.5.18	TEventLog.TimeStampFormat	151
10.5.19	TEventLog.CustomLogType	151
10.5.20	TEventLog.EventIDOffset	151
10.5.21	TEventLog.OnGetCustomCategory	152
10.5.22	TEventLog.OnGetCustomEventID	152
10.5.23	TEventLog.OnGetCustomEvent	152
<b>11</b>	<b>Reference for unit 'ezcgi'</b>	<b>153</b>
11.1	Used units	153
11.2	Overview	153
11.3	Constants, types and variables	153
11.3.1	Constants	153
11.4	ECGIException	153
11.4.1	Description	153
11.5	TEZcgi	154
11.5.1	Description	154
11.5.2	Method overview	154
11.5.3	Property overview	154
11.5.4	TEZcgi.Create	154
11.5.5	TEZcgi.Destroy	154
11.5.6	TEZcgi.Run	155
11.5.7	TEZcgi.WriteContent	155
11.5.8	TEZcgi.PutLine	155
11.5.9	TEZcgi.GetValue	156
11.5.10	TEZcgi.DoPost	156
11.5.11	TEZcgi.DoGet	156
11.5.12	TEZcgi.Values	156
11.5.13	TEZcgi.Names	157

11.5.14 TEZcgi.Variables . . . . .	157
11.5.15 TEZcgi.VariableCount . . . . .	158
11.5.16 TEZcgi.Name . . . . .	158
11.5.17 TEZcgi.Email . . . . .	158
<b>12 Reference for unit 'gettext' . . . . .</b>	<b>159</b>
12.1 Used units . . . . .	159
12.2 Overview . . . . .	159
12.3 Constants, types and variables . . . . .	159
12.3.1 Constants . . . . .	159
12.3.2 Types . . . . .	159
12.4 Procedures and functions . . . . .	160
12.4.1 GetLanguageIDs . . . . .	160
12.4.2 TranslateResourceStrings . . . . .	161
12.4.3 TranslateUnitResourceStrings . . . . .	161
12.5 EMOFileError . . . . .	161
12.5.1 Description . . . . .	161
12.6 TMOFile . . . . .	161
12.6.1 Description . . . . .	161
12.6.2 Method overview . . . . .	162
12.6.3 TMOFile.Create . . . . .	162
12.6.4 TMOFile.Destroy . . . . .	162
12.6.5 TMOFile.Translate . . . . .	162
<b>13 Reference for unit 'idea' . . . . .</b>	<b>163</b>
13.1 Used units . . . . .	163
13.2 Overview . . . . .	163
13.3 Constants, types and variables . . . . .	163
13.3.1 Constants . . . . .	163
13.3.2 Types . . . . .	164
13.4 Procedures and functions . . . . .	164
13.4.1 CipherIdea . . . . .	164
13.4.2 DeKeyIdea . . . . .	164
13.4.3 EnKeyIdea . . . . .	165
13.5 EIDEAError . . . . .	165
13.5.1 Description . . . . .	165
13.6 TIDEADeCryptStream . . . . .	165
13.6.1 Description . . . . .	165
13.6.2 Method overview . . . . .	165
13.6.3 TIDEADeCryptStream.Read . . . . .	165
13.6.4 TIDEADeCryptStream.Seek . . . . .	166

13.7	TIDEAEncryptStream	166
13.7.1	Description	166
13.7.2	Method overview	166
13.7.3	TIDEAEncryptStream.Destroy	166
13.7.4	TIDEAEncryptStream.Write	167
13.7.5	TIDEAEncryptStream.Seek	167
13.7.6	TIDEAEncryptStream.Flush	167
13.8	TIDEAStream	168
13.8.1	Description	168
13.8.2	Method overview	168
13.8.3	Property overview	168
13.8.4	TIDEAStream.Create	168
13.8.5	TIDEAStream.Key	168
<b>14</b>	<b>Reference for unit 'inicol'</b>	<b>169</b>
14.1	Used units	169
14.2	Overview	169
14.3	Constants, types and variables	169
14.3.1	Constants	169
14.4	EIniCol	170
14.4.1	Description	170
14.5	TIniCollection	170
14.5.1	Description	170
14.5.2	Method overview	170
14.5.3	Property overview	170
14.5.4	TIniCollection.Load	170
14.5.5	TIniCollection.Save	171
14.5.6	TIniCollection.SaveToIni	171
14.5.7	TIniCollection.SaveToFile	171
14.5.8	TIniCollection.LoadFromIni	172
14.5.9	TIniCollection.LoadFromFile	172
14.5.10	TIniCollection.Prefix	172
14.5.11	TIniCollection.SectionPrefix	173
14.5.12	TIniCollection.FileName	173
14.5.13	TIniCollection.GlobalSection	173
14.6	TIniCollectionItem	173
14.6.1	Description	173
14.6.2	Method overview	174
14.6.3	Property overview	174
14.6.4	TIniCollectionItem.SaveToIni	174

14.6.5	TIniCollectionItem.LoadFromIni	174
14.6.6	TIniCollectionItem.SaveToFile	174
14.6.7	TIniCollectionItem.LoadFromFile	175
14.6.8	TIniCollectionItem.SectionName	175
14.7	TNamedIniCollection	175
14.7.1	Description	175
14.7.2	Method overview	175
14.7.3	Property overview	176
14.7.4	TNamedIniCollection.IndexOfUserData	176
14.7.5	TNamedIniCollection.IndexOfName	176
14.7.6	TNamedIniCollection.FindByName	176
14.7.7	TNamedIniCollection.FindByUserData	177
14.7.8	TNamedIniCollection.NamedItems	177
14.8	TNamedIniCollectionItem	177
14.8.1	Description	177
14.8.2	Property overview	177
14.8.3	TNamedIniCollectionItem.UserData	177
14.8.4	TNamedIniCollectionItem.Name	178
<b>15</b>	<b>Reference for unit 'IniFiles'</b>	<b>179</b>
15.1	Used units	179
15.2	Overview	179
15.3	TCustomIniFile	179
15.3.1	Description	179
15.3.2	Method overview	180
15.3.3	Property overview	180
15.3.4	TCustomIniFile.Create	180
15.3.5	TCustomIniFile.Destroy	181
15.3.6	TCustomIniFile.SectionExists	181
15.3.7	TCustomIniFile.ReadString	181
15.3.8	TCustomIniFile.WriteString	182
15.3.9	TCustomIniFile.ReadInteger	182
15.3.10	TCustomIniFile.WriteInteger	182
15.3.11	TCustomIniFile.ReadBool	182
15.3.12	TCustomIniFile.WriteBool	183
15.3.13	TCustomIniFile.ReadDate	183
15.3.14	TCustomIniFile.ReadDateTime	183
15.3.15	TCustomIniFile.ReadFloat	184
15.3.16	TCustomIniFile.ReadTime	184
15.3.17	TCustomIniFile.ReadBinaryStream	184

15.3.18 TCustomIniFile.WriteDate . . . . .	185
15.3.19 TCustomIniFile.WriteDateTime . . . . .	185
15.3.20 TCustomIniFile.WriteFloat . . . . .	185
15.3.21 TCustomIniFile.WriteTime . . . . .	186
15.3.22 TCustomIniFile.WriteBinaryStream . . . . .	186
15.3.23 TCustomIniFile.ReadSection . . . . .	186
15.3.24 TCustomIniFile.ReadSections . . . . .	187
15.3.25 TCustomIniFile.ReadSectionValues . . . . .	187
15.3.26 TCustomIniFile.EraseSection . . . . .	187
15.3.27 TCustomIniFile.DeleteKey . . . . .	187
15.3.28 TCustomIniFile.UpdateFile . . . . .	188
15.3.29 TCustomIniFile.ValueExists . . . . .	188
15.3.30 TCustomIniFile.FileName . . . . .	188
15.3.31 TCustomIniFile.EscapeLineFeeds . . . . .	189
15.3.32 TCustomIniFile.CaseSensitive . . . . .	189
15.3.33 TCustomIniFile.StripQuotes . . . . .	189
15.4 THashedStringList . . . . .	189
15.4.1 Description . . . . .	189
15.4.2 Method overview . . . . .	190
15.4.3 THashedStringList.Create . . . . .	190
15.4.4 THashedStringList.Destroy . . . . .	190
15.4.5 THashedStringList.IndexOf . . . . .	190
15.4.6 THashedStringList.IndexOfName . . . . .	190
15.5 TIniFile . . . . .	191
15.5.1 Description . . . . .	191
15.5.2 Method overview . . . . .	191
15.5.3 Property overview . . . . .	191
15.5.4 TIniFile.Create . . . . .	191
15.5.5 TIniFile.Destroy . . . . .	191
15.5.6 TIniFile.ReadString . . . . .	192
15.5.7 TIniFile.WriteString . . . . .	192
15.5.8 TIniFile.ReadSection . . . . .	192
15.5.9 TIniFile.ReadSectionRaw . . . . .	192
15.5.10 TIniFile.ReadSections . . . . .	193
15.5.11 TIniFile.ReadSectionValues . . . . .	193
15.5.12 TIniFile.EraseSection . . . . .	193
15.5.13 TIniFile.DeleteKey . . . . .	193
15.5.14 TIniFile.UpdateFile . . . . .	194
15.5.15 TIniFile.Stream . . . . .	194
15.5.16 TIniFile.CacheUpdates . . . . .	194

15.6	TIniFileKey	195
15.6.1	Description	195
15.6.2	Method overview	195
15.6.3	Property overview	195
15.6.4	TIniFileKey.Create	195
15.6.5	TIniFileKey.Ident	195
15.6.6	TIniFileKey.Value	195
15.7	TIniFileKeyList	196
15.7.1	Description	196
15.7.2	Method overview	196
15.7.3	Property overview	196
15.7.4	TIniFileKeyList.Destroy	196
15.7.5	TIniFileKeyList.Clear	196
15.7.6	TIniFileKeyList.Items	196
15.8	TIniFileSection	197
15.8.1	Description	197
15.8.2	Method overview	197
15.8.3	Property overview	197
15.8.4	TIniFileSection.Empty	197
15.8.5	TIniFileSection.Create	197
15.8.6	TIniFileSection.Destroy	197
15.8.7	TIniFileSection.Name	198
15.8.8	TIniFileSection.KeyList	198
15.9	TIniFileSectionList	198
15.9.1	Description	198
15.9.2	Method overview	198
15.9.3	Property overview	198
15.9.4	TIniFileSectionList.Destroy	199
15.9.5	TIniFileSectionList.Clear	199
15.9.6	TIniFileSectionList.Items	199
15.10	TMemIniFile	199
15.10.1	Description	199
15.10.2	Method overview	199
15.10.3	TMemIniFile.Create	200
15.10.4	TMemIniFile.Clear	200
15.10.5	TMemIniFile.GetStrings	200
15.10.6	TMemIniFile.Rename	200
15.10.7	TMemIniFile.SetStrings	201

## 16 Reference for unit 'iostream'

202

16.1	Used units	202
16.2	Overview	202
16.3	Constants, types and variables	202
16.3.1	Types	202
16.4	EIOStreamError	203
16.4.1	Description	203
16.5	TIOStream	203
16.5.1	Description	203
16.5.2	Method overview	203
16.5.3	TIOStream.Create	203
16.5.4	TIOStream.Read	203
16.5.5	TIOStream.Write	204
16.5.6	TIOStream.SetSize	204
16.5.7	TIOStream.Seek	204
<b>17</b>	<b>Reference for unit 'Pipes'</b>	<b>205</b>
17.1	Used units	205
17.2	Overview	205
17.3	Constants, types and variables	205
17.3.1	Constants	205
17.4	Procedures and functions	205
17.4.1	CreatePipeHandles	205
17.4.2	CreatePipeStreams	206
17.5	EPipeCreation	206
17.5.1	Description	206
17.6	EPipeError	206
17.6.1	Description	206
17.7	EPipeSeek	206
17.7.1	Description	206
17.8	TInputPipeStream	206
17.8.1	Description	206
17.8.2	Method overview	207
17.8.3	Property overview	207
17.8.4	TInputPipeStream.Write	207
17.8.5	TInputPipeStream.Seek	207
17.8.6	TInputPipeStream.Read	208
17.8.7	TInputPipeStream.NumBytesAvailable	208
17.9	TOutputPipeStream	208
17.9.1	Description	208
17.9.2	Method overview	208

17.9.3	TOutputPipeStream.Seek . . . . .	208
17.9.4	TOutputPipeStream.Read . . . . .	209
<b>18</b>	<b>Reference for unit 'pooledmm'</b>	<b>210</b>
18.1	Used units . . . . .	210
18.2	Overview . . . . .	210
18.3	Constants, types and variables . . . . .	210
18.3.1	Types . . . . .	210
18.4	TNonFreePooledMemManager . . . . .	211
18.4.1	Description . . . . .	211
18.4.2	Method overview . . . . .	211
18.4.3	Property overview . . . . .	211
18.4.4	TNonFreePooledMemManager.Clear . . . . .	211
18.4.5	TNonFreePooledMemManager.Create . . . . .	211
18.4.6	TNonFreePooledMemManager.Destroy . . . . .	212
18.4.7	TNonFreePooledMemManager.NewItem . . . . .	212
18.4.8	TNonFreePooledMemManager.EnumerateItems . . . . .	212
18.4.9	TNonFreePooledMemManager.ItemSize . . . . .	212
18.5	TPooledMemManager . . . . .	213
18.5.1	Description . . . . .	213
18.5.2	Method overview . . . . .	213
18.5.3	Property overview . . . . .	213
18.5.4	TPooledMemManager.Clear . . . . .	213
18.5.5	TPooledMemManager.Create . . . . .	213
18.5.6	TPooledMemManager.Destroy . . . . .	213
18.5.7	TPooledMemManager.MinimumFreeCount . . . . .	214
18.5.8	TPooledMemManager.MaximumFreeCountRatio . . . . .	214
18.5.9	TPooledMemManager.Count . . . . .	214
18.5.10	TPooledMemManager.FreeCount . . . . .	215
18.5.11	TPooledMemManager.AllocatedCount . . . . .	215
18.5.12	TPooledMemManager.FreedCount . . . . .	215
<b>19</b>	<b>Reference for unit 'process'</b>	<b>216</b>
19.1	Used units . . . . .	216
19.2	Overview . . . . .	216
19.3	Constants, types and variables . . . . .	216
19.3.1	Types . . . . .	216
19.4	EProcess . . . . .	218
19.4.1	Description . . . . .	218
19.5	TProcess . . . . .	218
19.5.1	Description . . . . .	218

19.5.2	Method overview	219
19.5.3	Property overview	219
19.5.4	TProcess.Create	220
19.5.5	TProcess.Destroy	220
19.5.6	TProcess.Execute	220
19.5.7	TProcess.CloseInput	221
19.5.8	TProcess.CloseOutput	221
19.5.9	TProcess.CloseStderr	221
19.5.10	TProcess.Resume	221
19.5.11	TProcess.Suspend	222
19.5.12	TProcess.Terminate	222
19.5.13	TProcess.WaitOnExit	222
19.5.14	TProcess.WindowRect	223
19.5.15	TProcess.Handle	223
19.5.16	TProcess.ProcessHandle	223
19.5.17	TProcess.ThreadHandle	223
19.5.18	TProcess.ProcessID	224
19.5.19	TProcess.ThreadID	224
19.5.20	TProcess.Input	224
19.5.21	TProcess.Output	225
19.5.22	TProcess.Stderr	225
19.5.23	TProcess.ExitStatus	225
19.5.24	TProcess.InheritHandles	226
19.5.25	TProcess.Active	226
19.5.26	TProcess.ApplicationName	226
19.5.27	TProcess.CommandLine	226
19.5.28	TProcess.ConsoleTitle	227
19.5.29	TProcess.CurrentDirectory	227
19.5.30	TProcess.Desktop	227
19.5.31	TProcess.Environment	228
19.5.32	TProcess.Options	228
19.5.33	TProcess.Priority	229
19.5.34	TProcess.StartupOptions	229
19.5.35	TProcess.Running	230
19.5.36	TProcess.ShowWindow	230
19.5.37	TProcess.WindowColumns	231
19.5.38	TProcess.WindowHeight	231
19.5.39	TProcess.WindowLeft	231
19.5.40	TProcess.WindowRows	232
19.5.41	TProcess.WindowTop	232

19.5.42 TProcess.WindowWidth . . . . .	232
19.5.43 TProcess.FillAttribute . . . . .	233
<b>20 Reference for unit 'rttiutils' . . . . .</b>	<b>234</b>
20.1 Used units . . . . .	234
20.2 Overview . . . . .	234
20.3 Constants, types and variables . . . . .	234
20.3.1 Constants . . . . .	234
20.3.2 Types . . . . .	234
20.3.3 Variables . . . . .	235
20.4 Procedures and functions . . . . .	235
20.4.1 CreateStoredItem . . . . .	235
20.4.2 ParseStoredItem . . . . .	236
20.4.3 UpdateStoredList . . . . .	236
20.5 TPropInfoList . . . . .	236
20.5.1 Description . . . . .	236
20.5.2 Method overview . . . . .	236
20.5.3 Property overview . . . . .	237
20.5.4 TPropInfoList.Create . . . . .	237
20.5.5 TPropInfoList.Destroy . . . . .	237
20.5.6 TPropInfoList.Contains . . . . .	237
20.5.7 TPropInfoList.Find . . . . .	237
20.5.8 TPropInfoList.Delete . . . . .	238
20.5.9 TPropInfoList.Intersect . . . . .	238
20.5.10 TPropInfoList.Count . . . . .	238
20.5.11 TPropInfoList.Items . . . . .	238
20.6 TPropsStorage . . . . .	239
20.6.1 Description . . . . .	239
20.6.2 Method overview . . . . .	239
20.6.3 Property overview . . . . .	239
20.6.4 TPropsStorage.StoreAnyProperty . . . . .	239
20.6.5 TPropsStorage.LoadAnyProperty . . . . .	239
20.6.6 TPropsStorage.StoreProperties . . . . .	240
20.6.7 TPropsStorage.LoadProperties . . . . .	240
20.6.8 TPropsStorage.LoadObjectsProps . . . . .	240
20.6.9 TPropsStorage.StoreObjectsProps . . . . .	241
20.6.10 TPropsStorage.AObject . . . . .	242
20.6.11 TPropsStorage.Prefix . . . . .	242
20.6.12 TPropsStorage.Section . . . . .	242
20.6.13 TPropsStorage.OnReadString . . . . .	242

20.6.14 TPropsStorage.OnWriteString . . . . .	243
20.6.15 TPropsStorage.OnEraseSection . . . . .	243
<b>21 Reference for unit 'simpleipc' . . . . .</b>	<b>244</b>
21.1 Used units . . . . .	244
21.2 Overview . . . . .	244
21.3 Constants, types and variables . . . . .	244
21.3.1 Resource strings . . . . .	244
21.3.2 Constants . . . . .	245
21.3.3 Types . . . . .	245
21.3.4 Variables . . . . .	245
21.4 EIPCErrors . . . . .	246
21.4.1 Description . . . . .	246
21.5 TIPCCClientComm . . . . .	246
21.5.1 Description . . . . .	246
21.5.2 Method overview . . . . .	246
21.5.3 Property overview . . . . .	246
21.5.4 TIPCCClientComm.Create . . . . .	246
21.5.5 TIPCCClientComm.Connect . . . . .	246
21.5.6 TIPCCClientComm.Disconnect . . . . .	247
21.5.7 TIPCCClientComm.ServerRunning . . . . .	247
21.5.8 TIPCCClientComm.SendMessage . . . . .	247
21.5.9 TIPCCClientComm.Owner . . . . .	248
21.6 TIPCServerComm . . . . .	248
21.6.1 Description . . . . .	248
21.6.2 Method overview . . . . .	248
21.6.3 Property overview . . . . .	248
21.6.4 TIPCServerComm.Create . . . . .	248
21.6.5 TIPCServerComm.StartServer . . . . .	249
21.6.6 TIPCServerComm.StopServer . . . . .	249
21.6.7 TIPCServerComm.PeekMessage . . . . .	249
21.6.8 TIPCServerComm.ReadMessage . . . . .	250
21.6.9 TIPCServerComm.Owner . . . . .	250
21.6.10 TIPCServerComm.InstanceID . . . . .	250
21.7 TSimpleIPC . . . . .	250
21.7.1 Description . . . . .	250
21.7.2 Property overview . . . . .	250
21.7.3 TSimpleIPC.Active . . . . .	251
21.7.4 TSimpleIPC.ServerID . . . . .	251
21.8 TSimpleIPCClient . . . . .	251

21.8.1	Description	251
21.8.2	Method overview	251
21.8.3	Property overview	252
21.8.4	TSimpleIPCClient.Create	252
21.8.5	TSimpleIPCClient.Destroy	252
21.8.6	TSimpleIPCClient.Connect	252
21.8.7	TSimpleIPCClient.Disconnect	253
21.8.8	TSimpleIPCClient.ServerRunning	253
21.8.9	TSimpleIPCClient.SendMessage	253
21.8.10	TSimpleIPCClient.SendStringMessage	253
21.8.11	TSimpleIPCClient.SendStringMessageFmt	254
21.8.12	TSimpleIPCClient.ServerInstance	254
21.9	TSimpleIPCServer	254
21.9.1	Description	254
21.9.2	Method overview	255
21.9.3	Property overview	255
21.9.4	TSimpleIPCServer.Create	255
21.9.5	TSimpleIPCServer.Destroy	255
21.9.6	TSimpleIPCServer.StartServer	255
21.9.7	TSimpleIPCServer.StopServer	256
21.9.8	TSimpleIPCServer.PeekMessage	256
21.9.9	TSimpleIPCServer.GetMessageData	256
21.9.10	TSimpleIPCServer.StringMessage	257
21.9.11	TSimpleIPCServer.MsgType	257
21.9.12	TSimpleIPCServer.MsgData	257
21.9.13	TSimpleIPCServer.InstanceID	257
21.9.14	TSimpleIPCServer.Global	258
21.9.15	TSimpleIPCServer.OnMessage	258
<b>22</b>	<b>Reference for unit 'streamcoll'</b>	<b>259</b>
22.1	Used units	259
22.2	Overview	259
22.3	Procedures and functions	259
22.3.1	ColReadBoolean	259
22.3.2	ColReadCurrency	260
22.3.3	ColReadDateTime	260
22.3.4	ColReadFloat	260
22.3.5	ColReadInteger	260
22.3.6	ColReadString	261
22.3.7	ColWriteBoolean	261

22.3.8	ColWriteCurrency	261
22.3.9	ColWriteDateTime	261
22.3.10	ColWriteFloat	262
22.3.11	ColWriteInteger	262
22.3.12	ColWriteString	262
22.4	EStreamColl	262
22.4.1	Description	262
22.5	TStreamCollection	262
22.5.1	Description	262
22.5.2	Method overview	263
22.5.3	Property overview	263
22.5.4	TStreamCollection.LoadFromStream	263
22.5.5	TStreamCollection.SaveToStream	263
22.5.6	TStreamCollection.Streaming	263
22.6	TStreamCollectionItem	264
22.6.1	Description	264
<b>23</b>	<b>Reference for unit 'streamex'</b>	<b>265</b>
23.1	Used units	265
23.2	Overview	265
23.3	TBidirBinaryObjectReader	265
23.3.1	Description	265
23.3.2	Property overview	265
23.3.3	TBidirBinaryObjectReader.Position	265
23.4	TBidirBinaryObjectWriter	266
23.4.1	Description	266
23.4.2	Property overview	266
23.4.3	TBidirBinaryObjectWriter.Position	266
23.5	TDelphiReader	266
23.5.1	Description	266
23.5.2	Method overview	266
23.5.3	Property overview	266
23.5.4	TDelphiReader.GetDriver	267
23.5.5	TDelphiReader.ReadStr	267
23.5.6	TDelphiReader.Read	267
23.5.7	TDelphiReader.Position	267
23.6	TDelphiWriter	267
23.6.1	Description	267
23.6.2	Method overview	268
23.6.3	Property overview	268

23.6.4	TDelphiWriter.GetDriver . . . . .	268
23.6.5	TDelphiWriter.FlushBuffer . . . . .	268
23.6.6	TDelphiWriter.Write . . . . .	268
23.6.7	TDelphiWriter.WriteStr . . . . .	268
23.6.8	TDelphiWriter.WriteValue . . . . .	269
23.6.9	TDelphiWriter.Position . . . . .	269
<b>24</b>	<b>Reference for unit 'StreamIO'</b>	<b>270</b>
24.1	Used units . . . . .	270
24.2	Overview . . . . .	270
24.3	Procedures and functions . . . . .	270
24.3.1	AssignStream . . . . .	270
24.3.2	GetStream . . . . .	271
<b>25</b>	<b>Reference for unit 'syncobjs'</b>	<b>272</b>
25.1	Used units . . . . .	272
25.2	Overview . . . . .	272
25.3	Constants, types and variables . . . . .	272
25.3.1	Constants . . . . .	272
25.3.2	Types . . . . .	272
25.4	TCriticalSection . . . . .	273
25.4.1	Description . . . . .	273
25.4.2	Method overview . . . . .	273
25.4.3	TCriticalSection.Acquire . . . . .	274
25.4.4	TCriticalSection.Release . . . . .	274
25.4.5	TCriticalSection.Enter . . . . .	274
25.4.6	TCriticalSection.Leave . . . . .	274
25.4.7	TCriticalSection.Create . . . . .	275
25.4.8	TCriticalSection.Destroy . . . . .	275
25.5	TEventObject . . . . .	275
25.5.1	Description . . . . .	275
25.5.2	Method overview . . . . .	275
25.5.3	Property overview . . . . .	275
25.5.4	TEventObject.Create . . . . .	276
25.5.5	TEventObject.destroy . . . . .	276
25.5.6	TEventObject.ResetEvent . . . . .	276
25.5.7	TEventObject.SetEvent . . . . .	276
25.5.8	TEventObject.WaitFor . . . . .	277
25.5.9	TEventObject.ManualReset . . . . .	277
25.6	THandleObject . . . . .	277
25.6.1	Description . . . . .	277

25.6.2	Method overview	277
25.6.3	Property overview	277
25.6.4	THandleObject.destroy	277
25.6.5	THandleObject.Handle	278
25.6.6	THandleObject.LastError	278
25.7	TSimpleEvent	278
25.7.1	Description	278
25.7.2	Method overview	278
25.7.3	TSimpleEvent.Create	278
25.8	TSynchroObject	279
25.8.1	Description	279
25.8.2	Method overview	279
25.8.3	TSynchroObject.Acquire	279
25.8.4	TSynchroObject.Release	279
<b>26</b>	<b>Reference for unit 'zstream'</b>	<b>280</b>
26.1	Used units	280
26.2	Overview	280
26.3	Constants, types and variables	280
26.3.1	Types	280
26.4	Ecompressionerror	281
26.4.1	Description	281
26.5	Edecompressionerror	281
26.5.1	Description	281
26.6	Egzfileerror	281
26.6.1	Description	281
26.7	Ezliberror	281
26.7.1	Description	281
26.8	Tcompressionstream	281
26.8.1	Description	281
26.8.2	Method overview	282
26.8.3	Tcompressionstream.create	282
26.8.4	Tcompressionstream.destroy	282
26.8.5	Tcompressionstream.write	282
26.8.6	Tcompressionstream.flush	283
26.8.7	Tcompressionstream.get_compressionrate	283
26.9	Tcustomzlibstream	283
26.9.1	Description	283
26.9.2	Method overview	283
26.9.3	Tcustomzlibstream.create	283

26.9.4 Tcustomzlibstream.destroy . . . . .	284
26.10Tdecompressionstream . . . . .	284
26.10.1 Description . . . . .	284
26.10.2 Method overview . . . . .	284
26.10.3 Tdecompressionstream.create . . . . .	284
26.10.4 Tdecompressionstream.destroy . . . . .	284
26.10.5 Tdecompressionstream.read . . . . .	285
26.10.6 Tdecompressionstream.seek . . . . .	285
26.10.7 Tdecompressionstream.get_compressionrate . . . . .	286
26.11TGZFileStream . . . . .	286
26.11.1 Description . . . . .	286
26.11.2 Method overview . . . . .	286
26.11.3 TGZFileStream.create . . . . .	286
26.11.4 TGZFileStream.read . . . . .	286
26.11.5 TGZFileStream.write . . . . .	287
26.11.6 TGZFileStream.seek . . . . .	287
26.11.7 TGZFileStream.destroy . . . . .	287

## 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 have 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 implement 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 'AVL\_Tree'

### 1.1 Used units

Table 1.1: Used units by unit 'AVL\_Tree'

Name	Page
Classes	??
sysutils	??

### 1.2 Overview

The `avl_tree` unit implements a general-purpose AVL (balanced) tree class: the `TAVLTree` ([31](#)) class and it's associated data node class `TAVLTreeNode` ([40](#)).

### 1.3 TAVLTree

#### 1.3.1 Description

`TAVLTree` maintains a balanced AVL tree. The tree consists of `TAVLTreeNode` ([40](#)) 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` ([39](#)) property.

### 1.3.2 Method overview

Page	Property	Description
<a href="#">36</a>	Add	Add a new node to the tree
<a href="#">37</a>	Clear	Clears the tree
<a href="#">38</a>	ConsistencyCheck	Check the consistency of the tree
<a href="#">39</a>	Create	Create a new instance of <code>TAVLTree</code>
<a href="#">36</a>	Delete	Delete a node from the tree
<a href="#">39</a>	Destroy	Destroy the <code>TAVLTree</code> instance
<a href="#">32</a>	Find	Find a data item in the tree.
<a href="#">34</a>	FindHighest	Find the highest (rightmost) node in the tree.
<a href="#">33</a>	FindKey	Find a data item in the tree using alternate compare mechanism
<a href="#">34</a>	FindLeftMost	Find the node most left to a specified data node
<a href="#">35</a>	FindLeftMostKey	Find the node most left to a specified key node
<a href="#">35</a>	FindLeftMostSameKey	Find the node most left to a specified node with the same data
<a href="#">33</a>	FindLowest	Find the lowest (leftmost) node in the tree.
<a href="#">34</a>	FindNearest	Find the node closest to the data in the tree
<a href="#">34</a>	FindPointer	Search for a data pointer
<a href="#">33</a>	FindPrecessor	
<a href="#">35</a>	FindRightMost	Find the node most right to a specified node
<a href="#">35</a>	FindRightMostKey	Find the node most right to a specified key node
<a href="#">36</a>	FindRightMostSameKey	Find the node most right of a specified node with the same data
<a href="#">33</a>	FindSuccessor	Find successor to node
<a href="#">38</a>	FreeAndClear	Clears the tree and frees nodes
<a href="#">38</a>	FreeAndDelete	Delete a node from the tree and destroy it
<a href="#">37</a>	MoveDataLeftMost	Move data to the nearest left element
<a href="#">37</a>	MoveDataRightMost	Move data to the nearest right element
<a href="#">36</a>	Remove	Remove a data item from the list.
<a href="#">37</a>	RemovePointer	Remove a pointer item from the list.
<a href="#">39</a>	ReportAsString	Return the tree report as a string
<a href="#">38</a>	WriteReportToStream	Write the contents of the tree consistency check to the stream

### 1.3.3 Property overview

Page	Property	Access	Description
<a href="#">40</a>	Count	r	Number of nodes in the tree.
<a href="#">39</a>	OnCompare	rw	Compare function used when comparing nodes

### 1.3.4 TAVLTree.Find

Synopsis: Find a data item in the tree.

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

Visibility: `public`

Description: `Find` uses the default `OnCompare` ([39](#)) 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` ([34](#)).

See also: `TAVLTree.OnCompare` ([39](#)), `TAVLTree.FindKey` ([33](#))

### 1.3.5 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` (39), `TAVLTree.Find` (32)

### 1.3.6 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` (33), `TAVLTree.MoveDataRightMost` (37)

### 1.3.7 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` (33), `TAVLTree.MoveDataLeftMost` (37)

### 1.3.8 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` (34)

### 1.3.9 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` (33)

### 1.3.10 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` (34), `TAVLTree.FindLowest` (33), `TAVLTree.Find` (32), `TAVLTree.FindKey` (33)

### 1.3.11 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` (32), where a custom compare function can be used.

The default `OnCompare` (39) compares the data pointers, so the default `Find` will return the same node as `FindPointer`

See also: `TAVLTree.Find` (32), `TAVLTree.FindKey` (33)

### 1.3.12 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` (35), `TAVLTree.FindLeftMostKey` (35), `TAVLTree.FindRightMostKey` (35)

### 1.3.13 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 succeeding 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` (34), `TAVLTree.FindLeftMostKey` (35), `TAVLTree.FindRightMostKey` (35)

### 1.3.14 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` (34), `TAVLTree.FindRightMost` (35), `TAVLTree.FindRightMostKey` (35)

### 1.3.15 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 succeeding node for `Key` and tries to move as far right in the tree as possible.

See also: `TAVLTree.FindLeftMost` (34), `TAVLTree.FindRightMost` (35), `TAVLTree.FindLeftMostKey` (35)

### 1.3.16 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: `FindLeftMostSameKey` finds the node most left from and with the same data as the specified node `ANode`.

See also: `TAVLTree.FindLeftMost` (34), `TAVLTree.FindLeftMostKey` (35), `TAVLTree.FindRightMostSameKey` (36)

### 1.3.17 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: `FindRightMostSameKey` finds the node most right from and with the same data as the specified node `ANode`.

See also: `TAVLTree.FindRightMost` (35), `TAVLTree.FindRightMostKey` (35), `TAVLTree.FindLeftMostSameKey` (35)

### 1.3.18 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` (36), `TAVLTree.Remove` (36)

### 1.3.19 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` (40) 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` (36), `TAVLTree.RemovePointer` (37), `TAVLTree.Clear` (37)

### 1.3.20 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` (32) and, if found, deletes it from the tree. Only the first occurrence of `Data` will be removed.

See also: `TAVLTree.Delete` (36), `TAVLTree.RemovePointer` (37), `TAVLTree.Clear` (37), `TAVLTree.Find` (32)

### 1.3.21 TAVLTree.RemovePointer

Synopsis: Remove a pointer item from the list.

Declaration: `procedure RemovePointer(Data: Pointer)`

Visibility: `public`

Description: `Remove` uses `FindPointer` (34) 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` (36), `TAVLTree.Delete` (36), `TAVLTree.Clear` (37)

### 1.3.22 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` (37)

### 1.3.23 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` (37)

### 1.3.24 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` (38) instead.

See also: `TAVLTree.FreeAndClear` (38), `TAVLTree.Delete` (36)

### 1.3.25 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` (37), `TAVLTree.Delete` (36), `TAVLTree.FreeAndDelete` (38)

### 1.3.26 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` (37), `TAVLTree.Delete` (36), `TAVLTree.FreeAndClear` (38)

### 1.3.27 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 contains 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` (38)

### 1.3.28 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 returned in `StreamSize`. This method is only useful for debugging purposes.

See also: `TAVLTree.ConsistencyCheck` (38)

### 1.3.29 TAVLTree.ReportAsString

Synopsis: Return the tree report as a string

Declaration: `function ReportAsString : String`

Visibility: public

Description: `ReportAsString` calls `WriteReportToStream` (38) and returns the stream data as a string.

See also: `TAVLTree.WriteReportToStream` (38)

### 1.3.30 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` (31). An alternate `OnCompare` (31) can be provided: the default `OnCompare` method compares the 2 data pointers of a node.

See also: `OnCompare` (31)

### 1.3.31 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` (39), `TAVLTree.Clean` (31)

### 1.3.32 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` (39)

### 1.3.33 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.

## 1.4 TAVLTreeNode

### 1.4.1 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.

### 1.4.2 Method overview

Page	Property	Description
<a href="#">40</a>	<code>Clear</code>	Clears the node's data
<a href="#">40</a>	<code>TreeDepth</code>	Level of the node in the tree below

### 1.4.3 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.

### 1.4.4 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 (??)`

## 1.5 TAVLTreeNodeMemManager

### 1.5.1 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.

### 1.5.2 Method overview

Page	Property	Description
<a href="#">41</a>	Clear	Frees all unused nodes
<a href="#">42</a>	Create	Create a new instance of <code>TAVLTreeNodeMemManager</code>
<a href="#">42</a>	Destroy	
<a href="#">41</a>	DisposeNode	Return a node to the free list
<a href="#">41</a>	NewNode	Create a new <code>TAVLTreeNode</code> instance

### 1.5.3 Property overview

Page	Property	Access	Description
<a href="#">43</a>	Count	r	Number of nodes in the list.
<a href="#">42</a>	MaximumFreeNodeRatio	rw	Maximum amount of free nodes in the list
<a href="#">42</a>	MinimumFreeNode	rw	Minimum amount of free nodes to be kept.

### 1.5.4 `TAVLTreeNodeMemManager.DisposeNode`

Synopsis: Return a node to the free list

Declaration: `procedure DisposeNode (ANode: TAVLTreeNode)`

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` ([41](#))

### 1.5.5 `TAVLTreeNodeMemManager.NewNode`

Synopsis: Create a new `TAVLTreeNode` instance

Declaration: `function NewNode : TAVLTreeNode`

Visibility: public

Description: `NewNode` returns a new `TAVLTreeNode` ([40](#)) instance. If there is a node in the free list, it are returned. If no more free nodes are present, a new node is created.

See also: `TAVLTreeNodeMemManager.DisposeNode` ([41](#))

### 1.5.6 `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` ([42](#)), `TAVLTreeNodeMemManager.MaximumFreeNodeRatio` ([42](#))

### 1.5.7 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 ([42](#))

### 1.5.8 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 ([42](#))

### 1.5.9 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 ([42](#))

### 1.5.10 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 ([43](#)), TAVLTreeNodeMemManager.MinimumFreeNode ([42](#))

### 1.5.11 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` ([42](#)), `TAVLTreeNodeMemManager.MaximumFreeNodeRatio` ([42](#))

## Chapter 2

# Reference for unit 'base64'

### 2.1 Used units

Table 2.1: Used units by unit 'base64'

Name	Page
Classes	??
sysutils	??

### 2.2 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` (47) for encoding, and `TBase64DecodingStream` (45) 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...

### 2.3 Constants, types and variables

#### 2.3.1 Types

```
TBase64DecodingMode = (bdmStrict, bdMIME)
```

Table 2.2: Enumeration values for type `TBase64DecodingMode`

Value	Explanation
<code>bdMIME</code>	MIME encoding
<code>bdmStrict</code>	Strict encoding

`TBase64DecodingMode` determines the decoding algorithm used by `TBase64DecodingStream` (45). 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` (45) 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.

## 2.4 EBase64DecodingException

### 2.4.1 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.

## 2.5 TBase64DecodingStream

### 2.5.1 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-seeking capability.

### 2.5.2 Method overview

Page	Property	Description
<a href="#">45</a>	Create	Create a new instance of the <code>TBase64DecodingStream</code> class
<a href="#">46</a>	Read	Read and decrypt data from the source stream
<a href="#">46</a>	Reset	Reset the stream
<a href="#">46</a>	Seek	Set stream position.

### 2.5.3 Property overview

Page	Property	Access	Description
<a href="#">47</a>	EOF	r	
<a href="#">47</a>	Mode	rw	Decoding mode

### 2.5.4 TBase64DecodingStream.Create

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

**Declaration:** `constructor Create(AInputStream: TStream)`  
`constructor Create(AInputStream: TStream; AMode: TBase64DecodingMode)`

**Visibility:** public

**Description:** `Create` creates a new instance of the `TBase64DecodingStream` class. It stores the source stream `AInputStream` 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 \(47\)](#), [TBase64DecodingMode \(44\)](#)

### 2.5.5 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 \(47\)](#), [TBase64DecodingStream.Read \(46\)](#)

### 2.5.6 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 \(45\)](#), [TBase64DecodingStream.Seek \(46\)](#), [#rtl.classes.TStream.Read \(??\)](#)

### 2.5.7 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 \(46\)](#), [TBase64DecodingStream.Write \(45\)](#), [TBase64EncodingStream.Seek \(48\)](#), [#rtl.classes.TStream.Seek \(??\)](#)

## 2.5.8 TBase64DecodingStream.EOF

Synopsis:

Declaration: Property EOF : Boolean

Visibility: public

Access: Read

Description:

## 2.5.9 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 ([45](#))

## 2.6 TBase64EncodingStream

### 2.6.1 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.

### 2.6.2 Method overview

Page	Property	Description
<a href="#">47</a>	Create	Create a new instance of the TBase64EncodingStream class.
<a href="#">48</a>	Destroy	Remove a TBase64EncodingStream instance from memory
<a href="#">48</a>	Seek	Position the stream
<a href="#">48</a>	Write	Write data to the stream.

### 2.6.3 TBase64EncodingStream.Create

Synopsis: Create a new instance of the TBase64EncodingStream class.

Declaration: constructor Create (AOutputStream: TStream)

Visibility: public

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

See also: TBase64EncodingStream.Destroy ([48](#)), TBase64DecodingStream.Create ([45](#))

### 2.6.4 TBase64EncodingStream.Destroy

Synopsis: Remove a TBase64EncodingStream instance 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 (47)

### 2.6.5 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 (48), TBase64EncodingStream.Read (47), TBase64DecodingStream.Write (45), #rtl.classes.TStream.Write (??)

### 2.6.6 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 don't change the current file pointer position. The encryption stream is not seekable.

Errors: An `EStreamError` error is raised.

See also: TBase64EncodingStream.Read (47), TBase64EncodingStream.Write (48), #rtl.classes.TStream.Seek (??)

## Chapter 3

# Reference for unit 'BlowFish'

### 3.1 Used units

Table 3.1: Used units by unit 'BlowFish'

Name	Page
Classes	??
sysutils	??

### 3.2 Overview

The BlowFish implements a class TBlowFish ([50](#)) to handle blowfish encryption/decryption of memory buffers, and 2 TStream (??) descendents TBlowFishDeCryptStream ([51](#)) which descrypts any data that is read from it on the fly, as well as TBlowFishEnCryptStream ([52](#)) which encrypts the data that is written to it on the fly.

### 3.3 Constants, types and variables

#### 3.3.1 Constants

`BFRounds = 16`

Number of rounds in blowfish encryption.

#### 3.3.2 Types

`PBlowFishKey = ^TBlowFishKey`

PBlowFishKey is a simple pointer to a TBlowFishKey ([50](#)) array.

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

TBFBlock is the basic data structure used by the encrypting/decrypting routines in TBlowFish (50), TBlowFishDeCryptStream (51) and TBlowFishEnCryptStream (52). 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 (50), TBlowFishDeCryptStream (51) and TBlowFishEnCryptStream (52) classes. It should be filled with the encryption key and passed to the constructor of one of these classes.

## 3.4 EBlowFishError

### 3.4.1 Description

EBlowFishError is used by the TBlowFishStream (53), TBlowFishEncryptStream (52) and TBlowFishDecryptStream (51) classes to report errors.

## 3.5 TBlowFish

### 3.5.1 Description

TBlowFish is a simple class that can be used to encrypt/decrypt a single TBFBlock (50) data block with the Encrypt (50) and Decrypt (51) calls. It is used internally by the TBlowFishEnCryptStream (52) and TBlowFishDeCryptStream (51) classes to encrypt or decrypt the actual data.

### 3.5.2 Method overview

Page	Property	Description
<a href="#">50</a>	Create	Create a new instance of the TBlowFish class
<a href="#">51</a>	Decrypt	Decrypt a block
<a href="#">50</a>	Encrypt	Encrypt a block

### 3.5.3 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 (50) and Decrypt (51).

See also: TBlowFish.Encrypt (50), TBlowFish.Decrypt (51)

### 3.5.4 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 (50) specified when the `TBlowFish` instance was created.

See also: `TBlowFishKey` (50), `TBlowFish.Decrypt` (51), `TBlowFish.Create` (50)

### 3.5.5 TBlowFish.Decrypt

Synopsis: Decrypt a block

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

Visibility: public

Description: `Decrypt` decrypts the data in `Block` (always 8 bytes) using the key (50) specified when the `TBlowFish` instance was created. The data must have been encrypted with the same key and the `Encrypt` (50) call.

See also: `TBlowFishKey` (50), `TBlowFish.Encrypt` (50), `TBlowFish.Create` (50)

## 3.6 TBlowFishDeCryptStream

### 3.6.1 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` (52) stream must be used.

### 3.6.2 Method overview

Page	Property	Description
51	Read	Read data from the stream
52	Seek	Set the stream position.

### 3.6.3 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` (54), `TBlowFishEncryptStream` (52)

### 3.6.4 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` (50) exception will be raised.

See also: `TBlowFishDecryptStream.Read` (51), `EBlowFishError` (50)

## 3.7 TBlowFishEncryptStream

### 3.7.1 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` (51) stream must be used.

### 3.7.2 Method overview

Page	Property	Description
<a href="#">52</a>	<code>Destroy</code>	Free the <code>TBlowFishEncryptStream</code>
<a href="#">53</a>	<code>Flush</code>	Flush the encryption buffer
<a href="#">53</a>	<code>Seek</code>	Set the position in the stream
<a href="#">52</a>	<code>Write</code>	Write data to the stream

### 3.7.3 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` (53), `TBlowFishStream.Create` (54)

### 3.7.4 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` (53) mechanism can be used to write the remaining bytes.

See also: `TBlowFishEncryptStream.Read` (52)

### 3.7.5 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` (50) exception.

See also: `TBlowFishEncryptStream.Write` (52)

### 3.7.6 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` (52), `TBFBlock` (50)

## 3.8 TBlowFishStream

### 3.8.1 Description

`TBlowFishStream` is an abstract class which is used as a parent class for `TBlowFishEncryptStream` (52) and `TBlowFishDecryptStream` (51). It simply provides a constructor and storage for a `TBlowFish` (50) 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`.

### 3.8.2 Method overview

Page	Property	Description
<a href="#">54</a>	Create	Create a new instance of the <code>TBlowFishStream</code> class
<a href="#">54</a>	Destroy	Destroy the <code>TBlowFishStream</code> instance.

### 3.8.3 Property overview

Page	Property	Access	Description
<a href="#">54</a>	BlowFish	r	Blowfish instance used when encrypting/decrypting

### 3.8.4 TBlowFishStream.Create

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

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

Visibility: `public`

Description: `Create` initializes a new instance of `TBlowFishStream`, and creates an internal instance of `TBlowFish` ([50](#)) 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`.

See also: `TBlowFishEncryptStream` ([52](#)), `TBlowFishDecryptStream` ([51](#)), `TBlowFish` ([50](#))

### 3.8.5 TBlowFishStream.Destroy

Synopsis: Destroy the `TBlowFishStream` instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` cleans up the internal `TBlowFish` ([50](#)) instance.

Errors:

See also: `TBlowFishStream.Create` ([54](#)), `TBlowFish` ([50](#))

### 3.8.6 TBlowFishStream.BlowFish

Synopsis: Blowfish instance used when encrypting/decrypting

Declaration: `Property BlowFish : TBlowFish`

Visibility: `public`

Access: `Read`

Description: `BlowFish` is the `TBlowFish` ([50](#)) instance which is created when the `TBlowFishStream` class is initialized. Normally it should not be used directly, it's intended for access by the descendent classes `TBlowFishEncryptStream` ([52](#)) and `TBlowFishDecryptStream` ([51](#)).

See also: `TBlowFishEncryptStream` ([52](#)), `TBlowFishDecryptStream` ([51](#)), `TBlowFish` ([50](#))

## Chapter 4

# Reference for unit 'bufstream'

### 4.1 Used units

Table 4.1: Used units by unit 'bufstream'

Name	Page
Classes	??
sysutils	??

### 4.2 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 (58) which is for reading only, and TWriteBufStream (58) 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.

### 4.3 Constants, types and variables

#### 4.3.1 Constants

`DefaultBufferCapacity : Integer = 16`

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

### 4.4 TBufStream

#### 4.4.1 Description

TBufStream is the common ancestor for the TReadBufStream (58) and TWriteBufStream (58) 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.

#### 4.4.2 Method overview

Page	Property	Description
<a href="#">56</a>	Create	Create a new <code>TBufStream</code> instance.
<a href="#">56</a>	Destroy	Destroys the <code>TBufStream</code> instance

#### 4.4.3 Property overview

Page	Property	Access	Description
<a href="#">56</a>	Buffer	r	The current buffer
<a href="#">57</a>	BufferPos	r	Current buffer position.
<a href="#">57</a>	BufferSize	r	Amount of data in the buffer
<a href="#">57</a>	Capacity	rw	Current buffer capacity

#### 4.4.4 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` ([55](#)) is used as the capacity.

An instance of `TBufStream` should never be instantiated directly. Instead, an instance of `TReadBufStream` ([58](#)) or `TWriteBufStream` ([58](#)) should be created.

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

See also: `TBufStream.Destroy` ([56](#)), `TReadBufStream` ([58](#)), `TWriteBufStream` ([58](#))

#### 4.4.5 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` ([56](#)), `TReadBufStream` ([58](#)), `TWriteBufStream` ([58](#))

#### 4.4.6 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` (56), `TBufStream.Capacity` (57), `TBufStream.BufferSize` (57)

#### 4.4.7 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` (56), `TBufStream.Buffer` (56), `TBufStream.BufferSize` (57), `TBufStream.BufferPos` (57)

#### 4.4.8 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` (56), `TBufStream.Buffer` (56), `TBufStream.BufferSize` (57), `TBufStream.Capacity` (57)

#### 4.4.9 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` (57).

See also: `TBufStream.Create` (56), `TBufStream.Buffer` (56), `TBufStream.BufferPos` (57), `TBufStream.Capacity` (57)

## 4.5 TReadBufStream

### 4.5.1 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.

### 4.5.2 Method overview

Page	Property	Description
<a href="#">58</a>	Read	Reads data from the stream
<a href="#">58</a>	Seek	Set location in the buffer

### 4.5.3 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` ([59](#)), `TReadBufStream.Read` ([58](#)), `TReadBufStream.Write` ([58](#))

### 4.5.4 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 until `ACount` bytes are read, or no more data is available.

See also: `TReadBufStream.Seek` ([58](#)), `TReadBufStream.Read` ([58](#))

## 4.6 TWriteBufStream

### 4.6.1 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.

### 4.6.2 Method overview

Page	Property	Description
<a href="#">59</a>	Destroy	Remove the <code>TWriteBufStream</code> instance from memory
<a href="#">59</a>	Seek	Set stream position.
<a href="#">59</a>	Write	Write data to the stream

### 4.6.3 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` ([56](#)).

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

See also: `TBufStream.Create` ([56](#)), `TBufStream.Destroy` ([56](#))

### 4.6.4 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` ([59](#)), `TWriteBufStream.Read` ([58](#)), `TReadBufStream.Seek` ([58](#))

### 4.6.5 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` ([59](#)), `TWriteBufStream.Read` ([58](#)), `TReadBufStream.Write` ([58](#))

## Chapter 5

# Reference for unit 'CacheCls'

### 5.1 Used units

Table 5.1: Used units by unit 'CacheCls'

Name	Page
sysutils	??

### 5.2 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 called `TCache`

### 5.3 Constants, types and variables

#### 5.3.1 Resource strings

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

Message shown when an invalid index is passed.

#### 5.3.2 Types

```
PCacheSlot = ^TCacheSlot
```

Pointer to `TCacheSlot` (61) record.

```
PCacheSlotArray = ^TCacheSlotArray
```

Pointer to `TCacheSlotArray` (61) array

```
TCacheSlot = record
```

```

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

```

TCacheSlot is internally used by the TCache (61) 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 (60) 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 (62) 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.

## 5.4 ECacheError

### 5.4.1 Description

Exception class used in the cachecls unit.

## 5.5 TCache

### 5.5.1 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 (64) property offers indexed access to the array of items. When accessing the array through this property, the MRUSlot (64) property is updated.

### 5.5.2 Method overview

Page	Property	Description
<a href="#">62</a>	Add	Add a data element to the list.
<a href="#">63</a>	AddNew	Add a new item to the list.
<a href="#">62</a>	Create	Create a new cache class.
<a href="#">62</a>	Destroy	Free the TCache class from memory
<a href="#">63</a>	FindSlot	Find data pointer in the list
<a href="#">63</a>	IndexOf	Return index of a data pointer in the list.
<a href="#">64</a>	Remove	Remove a data item from the list.

### 5.5.3 Property overview

Page	Property	Access	Description
<a href="#">64</a>	Data	rw	Indexed access to data items
<a href="#">65</a>	LRUSlot	r	Last used item
<a href="#">64</a>	MRUSlot	rw	Most recent item slot.
<a href="#">66</a>	OnFreeSlot	rw	Event called when a slot is freed
<a href="#">65</a>	OnIsDataEqual	rw	Event to compare 2 items.
<a href="#">65</a>	SlotCount	rw	Number of slots in the list
<a href="#">65</a>	Slots	r	Indexed array to the slots

### 5.5.4 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` ([65](#))

### 5.5.5 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` ([62](#))

### 5.5.6 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` (63) 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` (63), `TCache.FindSlot` (63), `TCache.IndexOf` (63), `TCache.Data` (64), `TCache.MRUSlot` (64)

### 5.5.7 `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` (62) 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` (62), `TCache.FindSlot` (63), `TCache.IndexOf` (63), `TCache.Data` (64), `TCache.MRUSlot` (64)

### 5.5.8 `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` (65) event must be set.

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

See also: `TCache.IndexOf` (63), `TCache.Add` (62), `TCache.OnIsDataEqual` (65)

### 5.5.9 `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` (65) event must be set.

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

See also: `TCache.FindSlot` (63), `TCache.Add` (62), `TCache.OnIsDataEqual` (65)

### 5.5.10 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` (63)

### 5.5.11 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` (64) will point to the accessed item. The access is both read and write.

The index is zero-based and can maximally be `SlotCount-1` (65). Providing an invalid index will result in an exception.

See also: `TCache.MRUSlot` (64)

### 5.5.12 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` (64) property, or when an item is added to the list with `Add` (62) or `AddNew` (63)

See also: `TCache.Add` (62), `TCache.AddNew` (63), `TCache.Data` (64), `TCache.LRUSlot` (65)

### 5.5.13 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` (62), `TCache.AddNew` (63), `TCache.Data` (64), `TCache.MRUSlot` (64)

### 5.5.14 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` (66) is called.

See also: `TCache.Create` (62), `TCache.Data` (64), `TCache.Slots` (65)

### 5.5.15 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` (65). Providing an invalid index will result in an exception.

See also: `TCache.Data` (64), `TCache.SlotCount` (65)

### 5.5.16 TCache.OnIsDataEqual

Synopsis: Event to compare 2 items.

Declaration: `Property OnIsDataEqual : TOnIsDataEqual`

Visibility: `public`

Access: `Read,Write`

**Description:** `OnIsDataEqual` is used by `FindSlot` (63) and `IndexOf` (63) to compare items when looking for a particular item. These functions are called by the `Add` (62) 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` (63), `TCache.IndexOf` (63), `TCache.Add` (62)

### 5.5.17 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` (62), `TCache.AddNew` (63), `TCache.SlotCount` (65)

## Chapter 6

# Reference for unit 'contnrs'

### 6.1 Used units

Table 6.1: Used units by unit 'contnrs'

Name	Page
Classes	??
sysutils	??

### 6.2 Overview

The contnrs implements various general-purpose classes:

**Stacks** Stack classes to push/pop pointers or objects

**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.

### 6.3 Constants, types and variables

#### 6.3.1 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.

### 6.3.2 Types

PBucket = ^TBucket

PHashItem = ^THashItem

PHashItem is a pointer type, pointing to the THashItem (69) record.

PHashItemList = ^THashItemList

PHashItemList is a pointer to the THashItemList (70). It's used in the TFPHashList (85) as a pointer to the memory area containing the hash item records.

PHashTable = ^THashTable

PHashTable is a pointer to the THashTable (70). It's used in the TFPHashList (85) as a pointer to the memory area containing the hash values.

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

TBucketArray = Array of TBucket

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

Table 6.2: Enumeration values for type TBucketListSizes

Value	Explanation
bl128	
bl16	
bl2	
bl256	
bl32	
bl4	
bl64	
bl8	

TBucketItemArray = Array of TBucketItem

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

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

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

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

TDataIteratorMethod is a callback prototype for the TDataHashTable.Iterate (67) 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 (68) type. It's used in the TFPHashList (85) 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 (68) class.

```
THTCustomNodeClass = Class of THTCustomNode
```

THTCustomNodeClass is used by THTCustomHashTable (67) 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 (67) 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 (109) 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 (109) 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 (79) method. It is called for each element in the hash table, with the string. If Continue is set to false, the iteration stops.

## 6.4 Procedures and functions

### 6.4.1 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` (79) hash class.  
It's Robert Sedgwick's "Algorithms in C" hash function.

**Errors:** None.

**See also:** `TFPCustomHashTable` (79)

## 6.5 EDuplicate

### 6.5.1 Description

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

## 6.6 EKeyNotFound

### 6.6.1 Description

Exception raised when a key is not found.

## 6.7 TBucketList

### 6.7.1 Method overview

Page	Property	Description
71	Create	

### 6.7.2 TBucketList.Create

**Declaration:** `constructor Create(ABuckets: TBucketListSizes)`

**Visibility:** public

## 6.8 TClassList

### 6.8.1 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.

### 6.8.2 Method overview

Page	Property	Description
<a href="#">72</a>	Add	Add a new class pointer to the list.
<a href="#">72</a>	Extract	Extract a class pointer from the list.
<a href="#">73</a>	First	Return first non-nil class pointer
<a href="#">73</a>	IndexOf	Search for a class pointer in the list.
<a href="#">74</a>	Insert	Insert a new class pointer in the list.
<a href="#">73</a>	Last	Return last non- <code>Nil</code> class pointer
<a href="#">72</a>	Remove	Remove a class pointer from the list.

### 6.8.3 Property overview

Page	Property	Access	Description
<a href="#">74</a>	Items	rw	Index based access to class pointers.

### 6.8.4 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` (??) behaviour, 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` ([72](#)), `#rtl.classes.tlist.add` (??)

### 6.8.5 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` ([72](#)), `#rtl.classes.Tlist.Extract` (??)

### 6.8.6 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` (72), `#rtl.classes.Tlist.Remove` (??)

### 6.8.7 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 its position if it was found, or `-1` if it was not found in the list.

**Errors:** None.

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

### 6.8.8 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` (73), `TClassList.Pack` (71)

### 6.8.9 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` (73), `TClassList.Pack` (71)

### 6.8.10 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 (72)`, `TClassList.Remove (72)`

### 6.8.11 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 (??)`

## 6.9 TComponentList

### 6.9.1 Description

`TComponentList` is a `TObjectList (115)` 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 (115)`, `TComponentList` removes any `TComponent` from the list if the `TComponent` instance was freed externally. It uses the `FreeNotification` mechanism for this.

### 6.9.2 Method overview

Page	Property	Description
<a href="#">75</a>	Add	Add a component to the list.
<a href="#">75</a>	Destroy	Destroys the instance
<a href="#">75</a>	Extract	Remove a component from the list without destroying it.
<a href="#">76</a>	First	First non-nil instance in the list.
<a href="#">76</a>	IndexOf	Search for an instance in the list
<a href="#">77</a>	Insert	Insert a new component in the list
<a href="#">76</a>	Last	Last non-nil instance in the list.
<a href="#">75</a>	Remove	Remove a component from the list, possibly destroying it.

### 6.9.3 Property overview

Page	Property	Access	Description
<a href="#">77</a>	Items	rw	Index-based access to the elements in the list.

### 6.9.4 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` (115), `#rtl.classes.TComponent` (??)

### 6.9.5 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: `TObjectList.Add` (67)

### 6.9.6 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` (115) 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` (75), `TObjectList.Extract` (117)

### 6.9.7 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` (115) so it only accepts `TComponent` descendents. It introduces no new behaviour.

Errors: None.

See also: `TComponentList.Extract` (75), `TObjectList.Remove` (117)

### 6.9.8 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` (117)

### 6.9.9 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` (76), `TObjectList.First` (118)

### 6.9.10 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` (76), `TObjectList.Last` (118)

### 6.9.11 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` (118), `TComponentList.Add` (75), `TComponentList.Remove` (75)

### 6.9.12 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` (119)

## 6.10 TCustomBucketList

### 6.10.1 Method overview

Page	Property	Description
<a href="#">78</a>	Add	
<a href="#">78</a>	Assign	
<a href="#">78</a>	Clear	
<a href="#">77</a>	Destroy	
<a href="#">78</a>	Exists	
<a href="#">78</a>	Find	
<a href="#">78</a>	ForEach	
<a href="#">78</a>	Remove	

### 6.10.2 Property overview

Page	Property	Access	Description
<a href="#">78</a>	Data	rw	

### 6.10.3 TCustomBucketList.Destroy

Declaration: `destructor Destroy; Override`

Visibility: public

#### **6.10.4 TCustomBucketList.Clear**

Declaration: procedure Clear

Visibility: public

#### **6.10.5 TCustomBucketList.Add**

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

Visibility: public

#### **6.10.6 TCustomBucketList.Assign**

Declaration: procedure Assign(AList: TCustomBucketList)

Visibility: public

#### **6.10.7 TCustomBucketList.Exists**

Declaration: function Exists(AItem: Pointer) : Boolean

Visibility: public

#### **6.10.8 TCustomBucketList.Find**

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

Visibility: public

#### **6.10.9 TCustomBucketList.ForEach**

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

Visibility: public

#### **6.10.10 TCustomBucketList.Remove**

Declaration: function Remove(AItem: Pointer) : Pointer

Visibility: public

#### **6.10.11 TCustomBucketList.Data**

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

Visibility: public

Access: Read, Write

## 6.11 TFPCustomHashTable

### 6.11.1 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` (70).

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.

### 6.11.2 Method overview

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

### 6.11.3 Property overview

Page	Property	Access	Description
83	<code>AVGChainLen</code>	r	Average chain length
82	<code>Count</code>	r	Number of items in the hash table.
84	<code>Density</code>	r	Number of filled slots
81	<code>HashFunction</code>	rw	Hash function currently in use
82	<code>HashTable</code>	r	Hash table instance
82	<code>HashTableSize</code>	rw	Size of the hash table
83	<code>LoadFactor</code>	r	Fraction of count versus size
83	<code>MaxChainLength</code>	r	Maximum chain length
84	<code>NumberOfCollisions</code>	r	Number of extra items
83	<code>VoidSlots</code>	r	Number of empty slots in the hash table.

### 6.11.4 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` (70)

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

**See also:** `TFPCustomHashTable.CreateWith` (80)

### 6.11.5 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 (79)

### 6.11.6 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 (80), TFPCustomHashTable.Create (79), TFPCustomHashTable.CreateWith (80), THTCustomNode.Data (112)

### 6.11.7 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 (82)

### 6.11.8 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 (80)

### 6.11.9 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` (81), `TFPCustomHashTable.Add` (79)

### 6.11.10 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` (112) 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` (81) method should be used instead.

Errors: None.

See also: `TFPCustomHashTable.Add` (79), `TFPCustomHashTable.Delete` (81)

### 6.11.11 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` (82), `TFPCustomHashTable.HashTableSize` (82), `TFPCustomHashTable.AVGChainLen` (83), `TFPCustomHashTable.MaxChainLength` (83)

### 6.11.12 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` (79). 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` (79), `TFPCustomHashTable.HashTableSize` (82)

### 6.11.13 `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` (81), `TFPCustomHashTable.HashTableSize` (82), `TFPCustomHashTable.AVGChainLen` (83), `TFPCustomHashTable.MaxChainLength` (83)

### 6.11.14 `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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.AVGChainLen` (83), `TFPCustomHashTable.MaxChainLength` (83), `TFPCustomHashTable.VoidSlots` (83), `TFPCustomHashTable.Density` (84)

### 6.11.15 `TFPCustomHashTable.HashTable`

**Synopsis:** Hash table instance

**Declaration:** `Property HashTable : TFPObjectList`

**Visibility:** public

**Access:** Read

**Description:** `TFPCustomHashTable` is the internal list object (`TFPObjectList` (103)) used for the hash table. Each element in this table is again a `TFPObjectList` (103) instance or `Nil`.

### 6.11.16 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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.AVGChainLen` (83), `TFPCustomHashTable.MaxChainLength` (83), `TFPCustomHashTable.LoadFactor` (83), `TFPCustomHashTable.Density` (84), `TFPCustomHashTable.NumberOfCollisions` (84)

### 6.11.17 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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.AVGChainLen` (83), `TFPCustomHashTable.MaxChainLength` (83), `TFPCustomHashTable.VoidSlots` (83), `TFPCustomHashTable.Density` (84), `TFPCustomHashTable.NumberOfCollisions` (84)

### 6.11.18 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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.LoadFactor` (83), `TFPCustomHashTable.MaxChainLength` (83), `TFPCustomHashTable.VoidSlots` (83), `TFPCustomHashTable.Density` (84), `TFPCustomHashTable.NumberOfCollisions` (84)

### 6.11.19 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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.LoadFactor` (83), `TFPCustomHashTable.AvgChainLength` (79), `TFPCustomHashTable.VoidSlots` (83), `TFPCustomHashTable.Density` (84), `TFPCustomHashTable.NumberOfCollisions` (84)

### 6.11.20 `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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.LoadFactor` (83), `TFPCustomHashTable.AvgChainLength` (79), `TFPCustomHashTable.VoidSlots` (83), `TFPCustomHashTable.Density` (84)

### 6.11.21 `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` (81), `TFPCustomHashTable.Count` (82), `TFPCustomHashTable.LoadFactor` (83), `TFPCustomHashTable.AvgChainLength` (79), `TFPCustomHashTable.VoidSlots` (83), `TFPCustomHashTable.Density` (84)

## 6.12 `TFPDataHashTable`

### 6.12.1 Description

`TFPDataHashTable` is a `TFPCustomHashTable` (79) 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` (101), or for string data, `TFPStringHashTable` (111) is more suitable. The data pointers are exposed with their keys through the `Items` (85) property.

### 6.12.2 Method overview

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

### 6.12.3 Property overview

Page	Property	Access	Description
<a href="#">85</a>	Items	rw	Key-based access to the items in the table

### 6.12.4 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 ([85](#))

### 6.12.5 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 ([111](#))

## 6.13 TFPHashList

### 6.13.1 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 ([79](#)) class can be used. If a hash class for objects is needed instead of pointers, the TFPHashObjectList ([95](#)) class can be used.

### 6.13.2 Method overview

Page	Property	Description
<a href="#">87</a>	Add	Add a new key/data pair to the list
<a href="#">87</a>	Clear	Clear the list
<a href="#">86</a>	Create	Create a new instance of the hashlist
<a href="#">88</a>	Delete	Delete an item from the list.
<a href="#">86</a>	Destroy	Removes an instance of the hashlist from the heap
<a href="#">88</a>	Error	Raise an error
<a href="#">88</a>	Expand	Expand the list
<a href="#">88</a>	Extract	Extract a pointer from the list
<a href="#">89</a>	Find	Find data associated with key
<a href="#">89</a>	FindIndexOf	Return index of named item.
<a href="#">89</a>	FindWithHash	Find first element with given name and hash value
<a href="#">91</a>	ForEachCall	Call a procedure for each element in the list
<a href="#">87</a>	HashOfIndex	Return the hash value of an item by index
<a href="#">89</a>	IndexOf	Return the index of the data pointer
<a href="#">87</a>	NameOfIndex	Returns the key name of an item by index
<a href="#">90</a>	Pack	Remove nil pointers from the list
<a href="#">90</a>	Remove	Remove first instance of a pointer
<a href="#">90</a>	Rename	Rename a key
<a href="#">90</a>	ShowStatistics	Return some statistics for the list.

### 6.13.3 Property overview

Page	Property	Access	Description
<a href="#">91</a>	Capacity	rw	Capacity of the list.
<a href="#">91</a>	Count	rw	Current number of elements in the list.
<a href="#">91</a>	Items	rw	Indexed array with pointers
<a href="#">92</a>	List	r	Low-level hash list
<a href="#">92</a>	Strs	r	Low-level memory area with strings.

### 6.13.4 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` ([86](#))

### 6.13.5 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 \(86\)](#), [TFPHashList.Clear \(87\)](#)

### 6.13.6 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 \(88\)](#), [TFPHashList.Remove \(90\)](#), [TFPHashList.Delete \(88\)](#)

### 6.13.7 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 \(88\)](#), [TFPHashList.Remove \(90\)](#), [TFPHashList.Delete \(88\)](#), [TFPHashList.Add \(87\)](#)

### 6.13.8 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 \(87\)](#), [TFPHashList.Find \(89\)](#), [TFPHashList.FindIndexOf \(89\)](#), [TFPHashList.FindWithHash \(89\)](#)

### 6.13.9 TFPHashList.HashOfIndex

Synopsis: Return the hash value 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` (85), `TFPHashList.Find` (89), `TFPHashList.FindIndexOf` (89), `TFPHashList.FindWithHash` (89)

### 6.13.10 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` (88)`TFPHashList.Remove` (90)`TFPHashList.Add` (87)

### 6.13.11 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.

### 6.13.12 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` (87)

### 6.13.13 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` (88), `TFPHashList.Remove` (90), `TFPHashList.Clear` (87)

### 6.13.14 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` (87), `TFPHashList.NameOfIndex` (87), `TFPHashList.Find` (89), `TFPHashList.FindIndexOf` (89), `TFPHashList.FindWithHash` (89)

### 6.13.15 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` (87), `TFPHashList.NameOfIndex` (87), `TFPHashList.IndexOf` (89), `TFPHashList.FindIndexOf` (89), `TFPHashList.FindWithHash` (89)

### 6.13.16 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` (87), `TFPHashList.NameOfIndex` (87), `TFPHashList.IndexOf` (89), `TFPHashList.Find` (89), `TFPHashList.FindWithHash` (89)

### 6.13.17 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` (87), `TFPHashList.NameOfIndex` (87), `TFPHashList.IndexOf` (89), `TFPHashList.Find` (89), `TFPHashList.FindIndexOf` (89)

### 6.13.18 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.

### 6.13.19 TFPHashList.Remove

Synopsis: Remove first instance of a pointer

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

Visibility: public

Description: `Remove` removes the first occurrence 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` (88), `TFPHashList.Clear` (87), `TFPHashList.Extract` (88)

### 6.13.20 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` (87)

### 6.13.21 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:

**HashSize** Size of the hash table

**HashMean** Mean hash value

**HashStdDev** Standard deviation of hash values

**ListSize** Size and capacity of the list

**StringSize** Size and capacity of key strings

### 6.13.22 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`.

### 6.13.23 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` (91), `TFPHashList.Items` (91)

### 6.13.24 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` (91), `TFPHashList.Items` (91)

### 6.13.25 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` (91).

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

See also: `TFPHashList.Capacity` (91), `TFPHashList.Count` (91)

### 6.13.26 TFPHashList.List

Synopsis: Low-level hash list

Declaration: `Property List : PHashItemList`

Visibility: public

Access: Read

Description: `List` exposes the low-level item list (70). It should not be used directly.

See also: `TFPHashList.Strs` (92), `THashItemList` (70)

### 6.13.27 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` (92)

## 6.14 TFPHashObject

### 6.14.1 Description

`TFPHashObject` is a `TObject` descendent which is aware of the `TFPHashObjectList` (95) 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.

### 6.14.2 Method overview

Page	Property	Description
<a href="#">93</a>	<code>ChangeOwner</code>	Change the list owning the object.
<a href="#">93</a>	<code>ChangeOwnerAndName</code>	Simultaneously change the list owning the object and the name of the object.
<a href="#">93</a>	<code>Create</code>	Create a named instance, and insert in a hash list.
<a href="#">93</a>	<code>CreateNotOwned</code>	Create an instance not owned by any list.
<a href="#">94</a>	<code>Rename</code>	Rename the object

### 6.14.3 Property overview

Page	Property	Access	Description
<a href="#">94</a>	<code>Hash</code>	r	Hash value
<a href="#">94</a>	<code>Name</code>	r	Current name of the object

#### 6.14.4 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` (95) hash list. It also has no name when created in this way.

See also: `TFPHashObject.Name` (94), `TFPHashObject.ChangeOwner` (93), `TFPHashObject.ChangeOwnerAndName` (93)

#### 6.14.5 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` (95).

See also: `TFPHashObject.CreateNotOwned` (93), `TFPHashObject.ChangeOwner` (93), `TFPHashObject.Name` (94)

#### 6.14.6 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` (93), `TFPHashObject.Name` (94)

#### 6.14.7 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` (93), `TFPHashObject.Name` (94)

### 6.14.8 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` (93), `TFPHashObject.ChangeOwnerAndName` (93), `TFPHashObject.Name` (94)

### 6.14.9 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` (94), `TFPHashObject.ChangeOwnerAndName` (93)

### 6.14.10 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` (94)

## 6.15 TFPHashObjectList

### 6.15.1 Method overview

Page	Property	Description
<a href="#">96</a>	Add	Add a new key/data pair to the list
<a href="#">96</a>	Clear	Clear the list
<a href="#">95</a>	Create	Create a new instance of the hashlist
<a href="#">97</a>	Delete	Delete an object from the list.
<a href="#">95</a>	Destroy	Removes an instance of the hashlist from the heap
<a href="#">97</a>	Expand	Expand the list
<a href="#">97</a>	Extract	Extract a object instance from the list
<a href="#">98</a>	Find	Find data associated with key
<a href="#">98</a>	FindIndexOf	Return index of named object.
<a href="#">99</a>	FindInstanceOf	Search an instance of a certain class
<a href="#">99</a>	FindWithHash	Find first element with given name and hash value
<a href="#">100</a>	ForEachCall	Call a procedure for each object in the list
<a href="#">97</a>	HashOfIndex	Return the hash valye of an object by index
<a href="#">98</a>	IndexOf	Return the index of the object instance
<a href="#">96</a>	NameOfIndex	Returns the key name of an object by index
<a href="#">99</a>	Pack	Remove nil object instances from the list
<a href="#">98</a>	Remove	Remove first occurrence of a object instance
<a href="#">99</a>	Rename	Rename a key
<a href="#">100</a>	ShowStatistics	Return some statistics for the list.

### 6.15.2 Property overview

Page	Property	Access	Description
<a href="#">100</a>	Capacity	rw	Capacity of the list.
<a href="#">100</a>	Count	rw	Current number of elements in the list.
<a href="#">101</a>	Items	rw	Indexed array with object instances
<a href="#">101</a>	List	r	Low-level hash list
<a href="#">101</a>	OwnsObjects	rw	Does the list own the objects it contains

### 6.15.3 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` ([95](#)), `TFPHashObjectList.OwnsObjects` ([101](#))

### 6.15.4 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` (95), `TFPHashObjectList.Clear` (96)

### 6.15.5 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` (101) is `True`. It always frees all memory needed to contain the objects.

Errors: None.

See also: `TFPHashObjectList.Extract` (97), `TFPHashObjectList.Remove` (98), `TFPHashObjectList.Delete` (97), `TFPHashObjectList.Add` (96)

### 6.15.6 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` (97), `TFPHashObjectList.Remove` (98), `TFPHashObjectList.Delete` (97)

### 6.15.7 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` (97), `TFPHashObjectList.Find` (98), `TFPHashObjectList.FindIndexOf` (98), `TFPHashObjectList.FindWithHash` (99)

### 6.15.8 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` (95), `TFPHashObjectList.Find` (98), `TFPHashObjectList.FindIndexOf` (98), `TFPHashObjectList.FindWithHash` (99)

### 6.15.9 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` (101) is `True`, then the object itself is also freed from memory.

See also: `TFPHashObjectList.Extract` (97), `TFPHashObjectList.Remove` (98), `TFPHashObjectList.Add` (96), `TFPHashObjectList.OwnsObjects` (101)

### 6.15.10 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` (96)

### 6.15.11 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` (101).

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

See also: `TFPHashObjectList.Delete` (97), `TFPHashObjectList.Remove` (98), `TFPHashObjectList.Clear` (96)

### 6.15.12 TFPHashObjectList.Remove

Synopsis: Remove first occurrence of a object instance

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

Visibility: public

Description: `Remove` removes the first occurrence 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` (101) is `True`, then the object itself is also freed from memory.

See also: `TFPHashObjectList.Delete` (97), `TFPHashObjectList.Clear` (96), `TFPHashObjectList.Extract` (97)

### 6.15.13 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` (97), `TFPHashObjectList.NameOfIndex` (96), `TFPHashObjectList.Find` (98), `TFPHashObjectList.FindIndexOf` (98), `TFPHashObjectList.FindWithHash` (99)

### 6.15.14 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` (97), `TFPHashObjectList.NameOfIndex` (96), `TFPHashObjectList.IndexOf` (98), `TFPHashObjectList.FindIndexOf` (98), `TFPHashObjectList.FindWithHash` (99)

### 6.15.15 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` (97), `TFPHashObjectList.NameOfIndex` (96), `TFPHashObjectList.IndexOf` (98), `TFPHashObjectList.Find` (98), `TFPHashObjectList.FindWithHash` (99)

### 6.15.16 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` (97), `TFPHashObjectList.NameOfIndex` (96), `TFPHashObjectList.IndexOf` (98), `TFPHashObjectList.Find` (98), `TFPHashObjectList.FindIndexOf` (98)

### 6.15.17 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.

### 6.15.18 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.

### 6.15.19 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` (96)

### 6.15.20 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:

**HashSize**Size of the hash table

**HashMean**Mean hash value

**HashStdDev**Standard deviation of hash values

**ListSize**Size and capacity of the list

**StringSize**Size and capacity of key strings

### 6.15.21 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`.

### 6.15.22 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` ([100](#)), `TFPHashObjectList.Items` ([101](#))

### 6.15.23 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` ([100](#)), `TFPHashObjectList.Items` ([101](#))

### 6.15.24 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 \(95\)](#)

### 6.15.25 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` ([100](#)).

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

See also: [TFPHashObjectList.Capacity \(100\)](#), [TFPHashObjectList.Count \(100\)](#)

### 6.15.26 TFPHashObjectList.List

Synopsis: Low-level hash list

Declaration: `Property List : TFPHashList`

Visibility: `public`

Access: `Read`

Description: `List` exposes the low-level hash list ([85](#)). It should not be used directly.

See also: [TFPHashList \(85\)](#)

## 6.16 TFPObjectHashTable

### 6.16.1 Description

`TFPStringHashTable` is a `TFPCustomHashTable` ([79](#)) 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` ([111](#)), or for arbitrary pointer data, `TFPDataHashTable` ([84](#)) is more suitable. The objects are exposed with their keys through the `Items` ([103](#)) property.

### 6.16.2 Method overview

Page	Property	Description
<a href="#">103</a>	Add	Add a new object to the hash table
<a href="#">102</a>	Create	Create a new instance of TFPOjectHashTable
<a href="#">102</a>	CreateWith	Create a new hash table with given size and hash function

### 6.16.3 Property overview

Page	Property	Access	Description
<a href="#">103</a>	Items	rw	Key-based access to the objects
<a href="#">103</a>	OwnsObjects	rw	Does the hash table own the objects ?

### 6.16.4 TFPOjectHashTable.Create

Synopsis: Create a new instance of TFPOjectHashTable

Declaration: constructor Create(AOwnsObjects: Boolean)

Visibility: public

Description: Create creates a new instance of TFPOjectHashTable on the heap. It sets the OwnsObjects ([103](#)) 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: TFPOjectHashTable.OwnsObjects ([103](#)), TFPOjectHashTable.CreateWith ([102](#)), TFPOjectHashTable.Items ([103](#))

### 6.16.5 TFPOjectHashTable.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 ([103](#)) 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: TFPOjectHashTable.OwnsObjects ([103](#)), TFPOjectHashTable.Create ([102](#)), TFPOjectHashTable.Items ([103](#))

### 6.16.6 TFObjectHashTable.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: TFObjectHashTable.Items ([103](#))

### 6.16.7 TFObjectHashTable.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: TFObjectHashTable.Add ([103](#))

### 6.16.8 TFObjectHashTable.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: TFObjectHashTable.Create ([102](#)), TFObjectHashTable.Items ([103](#))

## 6.17 TFObjectList

### 6.17.1 Description

TFObjectList 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 ([115](#)), TFObjectList 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).

### 6.17.2 Method overview

Page	Property	Description
<a href="#">105</a>	Add	Add an object to the list.
<a href="#">108</a>	Assign	Copy the contents of a list.
<a href="#">105</a>	Clear	Clear all elements in the list.
<a href="#">104</a>	Create	Create a new object list
<a href="#">105</a>	Delete	Delete an element from the list.
<a href="#">104</a>	Destroy	Clears the list and destroys the list instance
<a href="#">106</a>	Exchange	Exchange the location of two objects
<a href="#">106</a>	Expand	Expand the capacity of the list.
<a href="#">106</a>	Extract	Extract an object from the list
<a href="#">107</a>	FindInstanceOf	Search for an instance of a certain class
<a href="#">108</a>	First	Return the first non-nil object in the list
<a href="#">109</a>	ForEachCall	For each object in the list, call a method or procedure, passing it the object.
<a href="#">107</a>	IndexOf	Search for an object in the list
<a href="#">107</a>	Insert	Insert a new object in the list
<a href="#">108</a>	Last	Return the last non-nil object in the list.
<a href="#">108</a>	Move	Move an object to another location in the list.
<a href="#">109</a>	Pack	Remove all Nil references from the list
<a href="#">106</a>	Remove	Remove an item from the list.
<a href="#">109</a>	Sort	Sort the list of objects

### 6.17.3 Property overview

Page	Property	Access	Description
<a href="#">110</a>	Capacity	rw	Capacity of the list
<a href="#">110</a>	Count	rw	Number of elements in the list.
<a href="#">110</a>	Items	rw	Indexed access to the elements of the list.
<a href="#">111</a>	List	r	Internal list used to keep the objects.
<a href="#">110</a>	OwnsObjects	rw	Should the list free elements when they are removed.

### 6.17.4 TFObjectList.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: `TFObjectList.Destroy` ([104](#)), `TFObjectList.OwnsObjects` ([110](#)), `TObjectList` ([115](#))

### 6.17.5 TFObjectList.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` (110) is `True`.

See also: `TFPObjectList.OwnsObjects` (110), `TObjectList.Create` (116)

### 6.17.6 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` (110) is `True`.

See also: `TObjectList.Destroy` (115)

### 6.17.7 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` (110) 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` (110), `TFPObjectList.Delete` (105)

### 6.17.8 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` (110) is `True`, the object is also freed.

Errors: An access violation may occur when `OwnsObjects` (110) is `True` and either the object was freed externally, or when the same object is in the same list twice.

See also: `TFPObjectList.Remove` (67), `TFPObjectList.Extract` (106), `TFPObjectList.OwnsObjects` (110), `TFPObjectList.Add` (67), `TFPObjectList.Clear` (67)

### 6.17.9 TFObjectList.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: `TTFObjectList.Add` (67), `TTFObjectList.Delete` (67)

### 6.17.10 TFObjectList.Expand

Synopsis: Expand the capacity of the list.

Declaration: `function Expand : TFObjectList`

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: `TFObjectList.Pack` (109), `TFObjectList.Clear` (105), `#rtl.classes.tfplist.expand (??)`

### 6.17.11 TFObjectList.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: `TFObjectList.Pack` (109), `TFObjectList.Clear` (105), `TFObjectList.Remove` (106), `TFObjectList.Delete` (105)

### 6.17.12 TFObjectList.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` (110) 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` ([109](#)), `TFPObjectList.Clear` ([105](#)), `TFPObjectList.Delete` ([105](#)), `TFPObjectList.Extract` ([106](#))

### 6.17.13 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` ([110](#)), `TFPObjectList.Remove` ([106](#)), `TFPObjectList.Extract` ([106](#))

### 6.17.14 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` ([107](#))

### 6.17.15 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` ([105](#)), `TFPObjectList.Delete` ([105](#))

### 6.17.16 TFObjectList.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: `TFObjectList.Last` (108), `TFObjectList.Pack` (109)

### 6.17.17 TFObjectList.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: `TFObjectList.First` (108), `TFObjectList.Pack` (109)

### 6.17.18 TFObjectList.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` (106), 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: `TFObjectList.Exchange` (106), `TFObjectList.Delete` (105), `TFObjectList.Insert` (107)

### 6.17.19 TFObjectList.Assign

Synopsis: Copy the contents of a list.

Declaration: `procedure Assign (Obj: TFObjectList)`

Visibility: public

Description: `Assign` copies the contents of `Obj` if `Obj` is of type `TFObjectList`

Errors: None.

### 6.17.20 TFObjectList.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: `TFObjectList.First` ([108](#)), `TFObjectList.Last` ([108](#))

### 6.17.21 TFObjectList.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` ([??](#))

### 6.17.22 TFObjectList.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` ([70](#)), `TObjectListCallback` ([70](#))

**6.17.23 TFObjectList.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` (110).

See also: `TFObjectList.Count` (110)

**6.17.24 TFObjectList.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: `TFObjectList.Capacity` (110)

**6.17.25 TFObjectList.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: `TFObjectList.Create` (104), `TFObjectList.Delete` (105), `TFObjectList.Remove` (106), `TFObjectList.Clear` (105)

**6.17.26 TFObjectList.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: `TFObjectList.Count` (110)

### 6.17.27 TFObjectList.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` (??)

## 6.18 TFPStringHashTable

### 6.18.1 Description

`TFPStringHashTable` is a `TFPCustomHashTable` (79) 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` (101), or for arbitrary pointer data, `TFPDataHashTable` (84) is more suitable. The strings are exposed with their keys through the `Items` (111) property.

### 6.18.2 Method overview

Page	Property	Description
<a href="#">111</a>	<code>Add</code>	Add a new string to the hash list

### 6.18.3 Property overview

Page	Property	Access	Description
<a href="#">111</a>	<code>Items</code>	rw	Key based access to the strings in the hash table

### 6.18.4 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` (111)

### 6.18.5 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` ([111](#))

## 6.19 THTCustomNode

### 6.19.1 Description

`THTCustomNode` is used by the `TFPCustomHashTable` ([79](#)) class to store the keys and associated values.

### 6.19.2 Method overview

Page	Property	Description
<a href="#">112</a>	<code>CreateWith</code>	Create a new instance of <code>THTCustomNode</code>
<a href="#">112</a>	<code>HasKey</code>	Check whether this node matches the given key.

### 6.19.3 Property overview

Page	Property	Access	Description
<a href="#">113</a>	<code>Key</code>	<code>r</code>	Key value associated with this hash item.

### 6.19.4 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` ([67](#)) class when needed.

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

See also: `TFPHashTable` ([67](#))

### 6.19.5 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` ([113](#))

### 6.19.6 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` ([112](#))

## 6.20 THTDataNode

### 6.20.1 Description

`THTDataNode` is used by `TDataHashTable` ([67](#)) 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`

### 6.20.2 Property overview

Page	Property	Access	Description
<a href="#">113</a>	<code>Data</code>	rw	Data pointer

### 6.20.3 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.

## 6.21 THTObjectNode

### 6.21.1 Description

`THTObjectNode` is a `THTCustomNode` ([112](#)) descendent which holds the data in the `TFPObjectHashTable` ([101](#)) hash table. It exposes a data string.

It should not be necessary to use `THTObjectNode` directly, it's only for inner use by `TFPObjectHashTable`

### 6.21.2 Property overview

Page	Property	Access	Description
<a href="#">114</a>	<code>Data</code>	rw	Object instance

### 6.21.3 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` (103)

See also: `TFPObjectHashTable` (101), `TFPObjectHashTable.Items` (103), `THTOwnedObjectNode` (114)

## 6.22 THTOwnedObjectNode

### 6.22.1 Description

`THTOwnedObjectNode` is used instead of `THTObjectNode` (113) in case `TFPObjectHashTable` (101) owns it's objects. When this object is destroyed, the associated data object is also destroyed.

### 6.22.2 Method overview

Page	Property	Description
114	Destroy	Destroys the node and the object.

### 6.22.3 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` (114), `TFPObjectHashTable.OwnsObjects` (103)

## 6.23 THTStringNode

### 6.23.1 Description

`THTStringNode` is a `THTCustomNode` (112) descendent which holds the data in the `TFPStringHashTable` (111) hash table. It exposes a data string.

It should not be necessary to use `THTStringNode` directly, it's only for inner use by `TFPStringHashTable`

### 6.23.2 Property overview

Page	Property	Access	Description
115	Data	rw	String data

### 6.23.3 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` ([111](#))

See also: `TFPStringHashTable` ([111](#))

## 6.24 TObjectBucketList

### 6.24.1 Method overview

Page	Property	Description
<a href="#">115</a>	Add	
<a href="#">115</a>	Remove	

### 6.24.2 Property overview

Page	Property	Access	Description
<a href="#">115</a>	Data	rw	

### 6.24.3 TObjectBucketList.Add

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

Visibility: public

### 6.24.4 TObjectBucketList.Remove

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

Visibility: public

### 6.24.5 TObjectBucketList.Data

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

Visibility: public

Access: Read,Write

## 6.25 TObjectList

### 6.25.1 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` (103), `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.

### 6.25.2 Method overview

Page	Property	Description
<a href="#">116</a>	<code>Add</code>	Add an object to the list.
<a href="#">116</a>	<code>create</code>	Create a new object list.
<a href="#">117</a>	<code>Extract</code>	Extract an object from the list.
<a href="#">117</a>	<code>FindInstanceOf</code>	Search for an instance of a certain class
<a href="#">118</a>	<code>First</code>	Return the first non-nil object in the list
<a href="#">117</a>	<code>IndexOf</code>	Search for an object in the list
<a href="#">118</a>	<code>Insert</code>	Insert an object in the list.
<a href="#">118</a>	<code>Last</code>	Return the last non-nil object in the list.
<a href="#">117</a>	<code>Remove</code>	Remove (and possibly free) an element from the list.

### 6.25.3 Property overview

Page	Property	Access	Description
<a href="#">119</a>	<code>Items</code>	rw	Indexed access to the elements of the list.
<a href="#">119</a>	<code>OwnsObjects</code>	rw	Should the list free elements when they are removed.

### 6.25.4 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` (115), `TObjectList.OwnsObjects` (119), `TFPObjectList` (103)

### 6.25.5 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` (118), `#rtl.classes.TList.Delete` (??), `TObjectList.Extract` (117), `TObjectList.Remove` (117)

### 6.25.6 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` (117), `Extract` does not free the extracted element if `OwnsObjects` (119) 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` (117)

### 6.25.7 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` (119) 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` (117)

### 6.25.8 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` (117)

### 6.25.9 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` ([117](#))

### 6.25.10 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` ([116](#)), `TObjectList.Remove` ([117](#))

### 6.25.11 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` ([118](#)), `TObjectList.Pack` ([115](#))

### 6.25.12 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` ([118](#)), `TObjectList.Pack` ([115](#))

### 6.25.13 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` (116), `TObjectList.Delete` (115), `TObjectList.Remove` (117), `TObjectList.Clear` (115)

### 6.25.14 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` (??)

## 6.26 TObjectQueue

### 6.26.1 Method overview

Page	Property	Description
<a href="#">120</a>	<code>Peek</code>	Look at the first object in the queue.
<a href="#">120</a>	<code>Pop</code>	Pop the first element off the queue
<a href="#">119</a>	<code>Push</code>	Push an object on the queue

### 6.26.2 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` (120), `TObjectQueue.Peek` (120)

### 6.26.3 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` ([119](#)), `TObjectQueue.Peek` ([120](#))

### 6.26.4 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` ([119](#)), `TObjectQueue.Pop` ([120](#))

## 6.27 TObjectStack

### 6.27.1 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.

### 6.27.2 Method overview

Page	Property	Description
<a href="#">121</a>	<code>Peek</code>	Look at the top object in the stack.
<a href="#">121</a>	<code>Pop</code>	Pop the top object of the stack.
<a href="#">120</a>	<code>Push</code>	Push an object on the stack.

### 6.27.3 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` ([121](#)), `TObjectStack.Peek` ([121](#))

### 6.27.4 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` ([120](#)), `TObjectStack.Peek` ([121](#))

### 6.27.5 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` ([120](#)), `TObjectStack.Pop` ([121](#))

## 6.28 TOrderedList

### 6.28.1 Description

`TOrderedList` provides the base class for `TQueue` ([123](#)) and `TStack` ([124](#)). 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.

### 6.28.2 Method overview

Page	Property	Description
<a href="#">122</a>	<code>AtLeast</code>	Check whether the list contains a certain number of elements.
<a href="#">122</a>	<code>Count</code>	Number of elements on the list.
<a href="#">121</a>	<code>Create</code>	Create a new ordered list
<a href="#">122</a>	<code>Destroy</code>	Free an ordered list
<a href="#">123</a>	<code>Peek</code>	Return the next element to be popped from the list.
<a href="#">123</a>	<code>Pop</code>	Remove an element from the list.
<a href="#">123</a>	<code>Push</code>	Push another element on the list.

### 6.28.3 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` ([122](#))

#### 6.28.4 `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` ([121](#))

#### 6.28.5 `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` ([122](#))

#### 6.28.6 `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` ([122](#))

### 6.28.7 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` ([123](#)), `TOrderedList.Peek` ([123](#))

### 6.28.8 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` ([123](#)), `TOrderedList.Push` ([123](#))

### 6.28.9 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` ([123](#)), without actually popping it from the list.

Errors: None.

See also: `TOrderedList.Pop` ([123](#)), `TOrderedList.Push` ([123](#))

## 6.29 TQueue

### 6.29.1 Description

`TQueue` is a descendent of `TOrderedList` ([121](#)) which implements `Push` ([123](#)) and `Pop` ([123](#)) 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` ([121](#))

## 6.30 TStack

### 6.30.1 Description

TStack is a descendent of TOrderedList (121) which implements Push (123) and Pop (123) 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 (121)

## Chapter 7

# Reference for unit 'CustApp'

### 7.1 Used units

Table 7.1: Used units by unit 'CustApp'

Name	Page
Classes	??
sysutils	??

### 7.2 Overview

The `CustApp` unit implements the `TCustomApplication` (126) 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.

### 7.3 Constants, types and variables

#### 7.3.1 Types

`TExceptionEvent` = procedure(Sender: TObject; E: Exception) of object

`TExceptionEvent` is the prototype for the exception handling events in `TCustomApplication`.

## 7.4 TCustomApplication

### 7.4.1 Description

`TCustomApplication` is the ancestor class for classes that wish to implement a global application class instance. It introduces several application-wide functionalities.

- Exception handling in `HandleException` (127), `ShowException` (128), `OnException` (132) and `StopOnException` (134).
- Command-line parameter parsing in `FindOptionIndex` (128), `GetOptionValue` (129), `CheckOptions` (130) and `HasOption` (129)
- Environment variable handling in `GetEnvironmentList` (131) and `EnvironmentVariable` (133).

Descendent classes need to override the `DoRun` protected method to implement the functionality of the program.

### 7.4.2 Method overview

Page	Property	Description
130	<code>CheckOptions</code>	Check whether all given options on the command-line are valid.
126	<code>Create</code>	Create a new instance of the <code>TCustomApplication</code> class
127	<code>Destroy</code>	Destroys the <code>TCustomApplication</code> instance.
128	<code>FindOptionIndex</code>	Return the index of an option.
131	<code>GetEnvironmentList</code>	Return a list of environment variables.
129	<code>GetOptionValue</code>	Return the value of a command-line option.
127	<code>HandleException</code>	Handle an exception.
129	<code>HasOption</code>	Check whether an option was specified.
127	<code>Initialize</code>	Initialize the application
128	<code>Run</code>	Runs the application.
128	<code>ShowException</code>	Show an exception to the user
128	<code>Terminate</code>	Terminate the application.

### 7.4.3 Property overview

Page	Property	Access	Description
134	<code>CaseSensitiveOptions</code>	rw	Are options interpreted case sensitive or not
132	<code>ConsoleApplication</code>	r	Is the application a console application or not
133	<code>EnvironmentVariable</code>	r	Environment variable access
131	<code>ExeName</code>	r	Name of the executable.
131	<code>HelpFile</code>	rw	Location of the application help file.
132	<code>Location</code>	r	Application location
132	<code>OnException</code>	rw	Exception handling event
134	<code>OptionChar</code>	rw	Command-line switch character
133	<code>ParamCount</code>	r	Number of command-line parameters
133	<code>Params</code>	r	Command-line parameters
134	<code>StopOnException</code>	rw	Should the program loop stop on an exception
131	<code>Terminated</code>	r	Was <code>Terminate</code> called or not
132	<code>Title</code>	rw	Application title

### 7.4.4 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` (127)

## 7.4.5 TCustomApplication.Destroy

**Synopsis:** Destroys the `TCustomApplication` instance.

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

**Visibility:** `public`

**Description:** `Destroy` simply calls the inherited `Destroy`.

See also: `TCustomApplication.Create` (126)

## 7.4.6 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` (132) 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` (128) 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` (134) property is set to `True`, the `Terminated` (131) property is set to `True`, which will cause the `Run` (128) loop to stop, and the application will exit.

See also: `TCustomApplication.ShowException` (128), `TCustomApplication.StopOnException` (134), `TCustomApplication.Terminated` (131), `TCustomApplication.Run` (128)

## 7.4.7 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` (128), `TCustomApplication.Terminated` (131)

### 7.4.8 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` (127). If `TCustomApplication.StopOnException` (134) 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` (127), `TCustomApplication.StopException` (126)

### 7.4.9 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` (127), `TCustomApplication.StopException` (126)

### 7.4.10 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` (131) property and properly shut down the application if it is set to `True`.

See also: `TCustomApplication.Terminated` (131), `TCustomApplication.Run` (128)

### 7.4.11 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` (134) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with `OptionChar` (134) (by default the dash ('-') character).

**See also:** `TCustomApplication.HasOption` (129), `TCustomApplication.GetOptionValue` (129), `TCustomApplication.CheckOptions` (130), `TCustomApplication.CaseSensitiveOptions` (134), `TCustomApplication.OptionChar` (134)

### 7.4.12 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` (134) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with `OptionChar` (134) (by default the dash ('-') character).

**See also:** `TCustomApplication.FindOptionIndex` (128), `TCustomApplication.HasOption` (129), `TCustomApplication.CheckOptions` (130), `TCustomApplication.CaseSensitiveOptions` (134), `TCustomApplication.OptionChar` (134)

### 7.4.13 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` (134) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with `OptionChar` (134) (by default the dash ('-') character).

**See also:** `TCustomApplication.FindOptionIndex` (128), `TCustomApplication.GetOptionValue` (129), `TCustomApplication.CheckOptions` (130), `TCustomApplication.CaseSensitiveOptions` (134), `TCustomApplication.OptionChar` (134)

#### 7.4.14 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 value 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` (134) (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` (128), `TCustomApplication.GetOptionValue` (129), `TCustomApplication.HasOption` (129), `TCustomApplication.CaseSensitiveOptions` (134), `TCustomApplication.OptionChar` (134)

### 7.4.15 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` ([133](#))

### 7.4.16 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` (??)

### 7.4.17 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` ([132](#))

### 7.4.18 TCustomApplication.Terminated

Synopsis: Was `Terminate` called or not

Declaration: `Property Terminated : Boolean`

Visibility: public

Access: Read

Description: `Terminated` indicates whether `Terminate` ([128](#)) 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` ([128](#))

### 7.4.19 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` ([131](#))

### 7.4.20 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` ([126](#)).

If set, `OnException` is called by the `HandleEvent` ([126](#)) routine. Do not use the `OnException` event directly, instead call `HandleEvent`

See also: `TCustomApplication.ShowEvent` ([126](#))

### 7.4.21 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` (??)

### 7.4.22 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` (133)

### 7.4.23 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` (133). It is equivalent to calling `ParamStr` (??).

See also: `TCustomApplication.ParamCount` (133), `#rtl.system.paramstr` (??)

### 7.4.24 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` (133) property.

See also: `TCustomApplication.Params` (133), `#rtl.system.paramstr` (??), `#rtl.system.paramcount` (??)

### 7.4.25 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` (131) can be used.

See also: `TCustomApplication.GetEnvironmentList` (131), `TCustomApplication.Params` (133)

### 7.4.26 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` (128), `TCustomApplication.GetOptionValue` (129), `TCustomApplication.HasOption` (129), `TCustomApplication.CaseSensitiveOptions` (134), `TCustomApplication.CheckOptions` (130)

### 7.4.27 TCustomApplication.CaseSensitiveOptions

Synopsis: Are options interpreted case sensitive or not

Declaration: `Property CaseSensitiveOptions : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `CaseSensitiveOptions` determines whether `FindOptionIndex` (128) and `CheckOptions` (130) 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` (128), `TCustomApplication.GetOptionValue` (129), `TCustomApplication.HasOption` (129), `TCustomApplication.OptionChar` (134), `TCustomApplication.CheckOptions` (130)

### 7.4.28 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` (128) and `HandleException` (127) procedures in case of an unhandled exception in the `DoRun` code. If `StopOnException` is `True` then `Terminate` (128) will be called after the exception was handled.

See also: `TCustomApplication.Run` (128), `TCustomApplication.HandleException` (127), `TCustomApplication.Terminate` (128)

## Chapter 8

# Reference for unit 'dbugintf'

### 8.1 Writing a debug server

Writing a debug server is relatively easy. It should instantiate a `TSimpleIPCTServer` class from the SimpleIPC (135) 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 (135) 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 (135) unit should also be functional.

### 8.2 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 (`dbugsrv`) 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 (135) 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 (135) unit should also be functional.

### 8.3 Constants, types and variables

#### 8.3.1 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.

### 8.3.2 Constants

`SendError : String = ''`

Whenever a call encounters an exception, the exception message is stored in this variable.

### 8.3.3 Types

`TDebugLevel = (dlInformation, dlWarning, dlError)`

Table 8.1: Enumeration values for type TDebugLevel

Value	Explanation
<code>dlError</code>	Error message
<code>dlInformation</code>	Informational message
<code>dlWarning</code>	Warning message

`TDebugLevel` indicates the severity level of the debug message to be sent. By default, an informational message is sent.

## 8.4 Procedures and functions

### 8.4.1 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` ([140](#)), `SendDebug` ([137](#))

### 8.4.2 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` (137) call will initialize the debug client when it is first called.

Errors: None.

See also: `SendDebug` (137), `StartDebugServer` (140)

### 8.4.3 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` (137) which sends the name and value of a boolean value as an informational message.

Errors: None.

See also: `SendDebug` (137), `SendDateTime` (137), `SendInteger` (138), `SendPointer` (139)

### 8.4.4 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` (137) which sends the name and value of an integer value as an informational message. The value is converted to a string using the `DateTimeToStr` (??) call.

Errors: None.

See also: `SendDebug` (137), `SendBoolean` (137), `SendInteger` (138), `SendPointer` (139)

### 8.4.5 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` (136).

See also: `SendDebugEx` (138), `SendDebugFmt` (138), `SendDebugFmtEx` (138)

### 8.4.6 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` (137) uses informational messages.

Other than that the function of `SendDebugEx` is equal to that of `SendDebug`

Errors: None.

See also: `SendDebug` (137), `SendDebugFmt` (138), `SendDebugFmtEx` (138)

### 8.4.7 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` (137). It exists mainly to avoid the `Format` call in calling code.

Errors: None.

See also: `SendDebug` (137), `SendDebugEx` (138), `SendDebugFmtEx` (138), `#rtl.sysutils.format` (??)

### 8.4.8 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` (138) with `Debug level MType`. It exists mainly to avoid the `Format` call in calling code.

Errors: None.

See also: `SendDebug` (137), `SendDebugEx` (138), `SendDebugFmt` (138), `#rtl.sysutils.format` (??)

### 8.4.9 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` (137) 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` (137), `SendBoolean` (137), `SendDateTime` (137), `SendPointer` (139)

#### 8.4.10 `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` (139), 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` (137), `SendMethodExit` (139), `SendSeparator` (140)

#### 8.4.11 `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` (139), 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` (137), `SendMethodEnter` (139), `SendSeparator` (140)

#### 8.4.12 `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` (137) 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` (137), `SendBoolean` (137), `SendDateTime` (137), `SendInteger` (138)

### 8.4.13 SendSeparator

Synopsis: Send a separator message

Declaration: `procedure SendSeparator`

Visibility: `default`

Description: `SendSeparator` is a simple wrapper around `SendDebug` (137) 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` (137), `SendMethodEnter` (139), `SendMethodExit` (139)

### 8.4.14 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` (137) 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` (136), `SendDebug` (137)

### 8.4.15 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` (137) call will attempt to start the server by itself if it is not yet running.

Errors: On error, `False` is returned.

See also: `SendDebug` (137), `InitDebugClient` (136)

## Chapter 9

# Reference for unit 'dbugmsg'

### 9.1 Used units

Table 9.1: Used units by unit 'dbugmsg'

Name	Page
Classes	??

### 9.2 Overview

dbugmsg is an auxiliary unit used in the dbugintf ([135](#)) unit. It defines the message protocol used between the debug unit and the debug server.

### 9.3 Constants, types and variables

#### 9.3.1 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.

### 9.3.2 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` (143) and `ReadDebugMessageFromStream` (142) can be used to read or write the message from/to a stream.

## 9.4 Procedures and functions

### 9.4.1 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` (142) 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.

### 9.4.2 ReadDebugMessageFromStream

**Synopsis:** Read a message from stream

**Declaration:** `procedure ReadDebugMessageFromStream(AStream: TStream;
var Msg: TDebugMessage)`

**Visibility:** default

**Description:** `ReadDebugMessageFromStream` reads a `TDebugMessage` (142) 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` (142), `WriteDebugMessageToStream` (143)

### 9.4.3 WriteDebugMessageToStream

Synopsis: Write a message to stream

Declaration: `procedure WriteDebugMessageToStream(AStream: TStream;  
const Msg: TDebugMessage)`

Visibility: default

Description: `WriteDebugMessageFromStream` writes a `TDebugMessage` (142) 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` (142), `ReadDebugMessageToStream` (141)

## Chapter 10

# Reference for unit 'eventlog'

### 10.1 Used units

Table 10.1: Used units by unit 'eventlog'

Name	Page
Classes	??
sysutils	??

### 10.2 Overview

The EventLog unit implements the TEventLog ([146](#)) 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.

### 10.3 Constants, types and variables

#### 10.3.1 Resource strings

SErrLogFailedMsg = 'Failed to log entry (Error: %s)'

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

### 10.3.2 Types

TEventType = (etCustom, etInfo, etWarning, etError, etDebug)

Table 10.2: 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 (152) event handler. It should return a OS event category 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 (152) and OnGetCustomEventID (152) 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 10.3: 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 (149) 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.

## 10.4 ELogError

### 10.4.1 Description

ELogError is the exception used in the TEventLog (146) component to indicate errors.

## 10.5 TEventLog

### 10.5.1 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 (148) call, or the specialized Warning (148), Error (148), Info (149) or Debug (149) calls, which have the event type predefined.

### 10.5.2 Method overview

Page	Property	Description
149	Debug	Log a debug message
146	Destroy	Clean up TEventLog instance
148	Error	Log an error message to
147	EventTypeToString	Create a string representation of an event type
149	Info	Log an informational message
148	Log	Log a message to the system log.
147	RegisterMessageFile	Register message file
148	Warning	Log a warning message.

### 10.5.3 Property overview

Page	Property	Access	Description
150	Active	rw	Activate the log mechanism
151	CustomLogType	rw	Custom log type ID
150	DefaultEventType	rw	Default event type for the Log (148) call.
151	EventIDOffset	rw	Offset for event ID messages identifiers
150	FileName	rw	File name for log file
149	Identification	rw	Identification string for messages
149	LogType	rw	Log type
152	OnGetCustomCategory	rw	Event to retrieve custom message category
152	OnGetCustomEvent	rw	Event to retrieve custom event Code
152	OnGetCustomEventID	rw	Event to retrieve custom event ID
150	RaiseExceptionOnError	rw	
151	TimeStampFormat	rw	Format for the timestamp string

### 10.5.4 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 (150) property to False.

See also: `TEventLog.Active` ([150](#))

### 10.5.5 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` ([145](#))

### 10.5.6 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` ([151](#)) (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.

### 10.5.7 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 `Default-EventType` (150).

If `EventType` is `etCustom`, then the `OnGetCustomEvent` (152), `OnGetCustomEventID` (152) and `OnGetCustomCategory` (152).

The other logging calls: `Info` (149), `Warning` (148), `Error` (148) and `Debug` (149) use the `Log` call to do the actual work.

See also: `TEventLog.Info` (149), `TEventLog.Warning` (148), `TEventLog.Error` (148), `TEventLog.Debug` (149), `TEventLog.OnGetCustomEvent` (152), `TEventLog.OnGetCustomEventID` (152), `TEventLog.OnGetCustomCategory` (152)

### 10.5.8 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` (148), `TEventLog.Info` (149), `TEventLog.Error` (148), `TEventLog.Debug` (149)

### 10.5.9 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` (148), `TEventLog.Info` (149), `TEventLog.Warning` (148), `TEventLog.Debug` (149)

### 10.5.10 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` ([148](#)), `TEventLog.Info` ([149](#)), `TEventLog.Warning` ([148](#)), `TEventLog.Error` ([148](#))

### 10.5.11 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` ([148](#)), `TEventLog.Warning` ([148](#)), `TEventLog.Error` ([148](#)), `TEventLog.Debug` ([149](#))

### 10.5.12 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` ([150](#)), `TEventLog.TimeStampFormat` ([151](#))

### 10.5.13 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` ([150](#)) property.

See also: `TEventLog.FileName` ([150](#))

### 10.5.14 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` (148)

### 10.5.15 TEventLog.RaiseExceptionOnError

Declaration: `Property RaiseExceptionOnError : Boolean`

Visibility: published

Access: Read,Write

### 10.5.16 TEventLog.DefaultEventType

Synopsis: Default event type for the `Log` (148) call.

Declaration: `Property DefaultEventType : TEventType`

Visibility: published

Access: Read,Write

Description: `DefaultEventType` is the event type used by the `Log` (148) call if it's `EventType` parameter is omitted.

See also: `TEventLog.Log` (148)

### 10.5.17 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` (145) 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` (145)

### 10.5.18 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` ([149](#))

### 10.5.19 TEventLog.CustomLogType

Synopsis: Custom log type ID

Declaration: `Property CustomLogType : Word`

Visibility: published

Access: Read,Write

Description: `CustomLogType` is used in the `EventTypeToString` ([147](#)) to format the custom log event type string.

See also: `TEventLog.EventTypeToString` ([147](#))

### 10.5.20 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` ([147](#)) 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` ([147](#))

### 10.5.21 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 ([152](#)), TEventLog.OnGetCustomEvent ([152](#))

### 10.5.22 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 ([152](#)), TEventLog.OnGetCustomEvent ([152](#))

### 10.5.23 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 ([152](#)), TEventLog.OnGetCustomEventID ([152](#))

# Chapter 11

## Reference for unit 'ezcgi'

### 11.1 Used units

Table 11.1: Used units by unit 'ezcgi'

Name	Page
Classes	??
strings	<a href="#">153</a>
sysutils	??

### 11.2 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` ([156](#)) or `DoGet` ([156](#)) methods should be overridden.

### 11.3 Constants, types and variables

#### 11.3.1 Constants

```
hexTable = '0123456789ABCDEF'
```

String constant used to convert a number to a hexadecimal code or back.

### 11.4 ECGIException

#### 11.4.1 Description

Exception raised by `TEZcgi` ([154](#))

## 11.5 TEZcgi

### 11.5.1 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 (156) or DoGet (156) methods. To run the program, an instance of this class must be created, and it's Run (155) 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.

### 11.5.2 Method overview

Page	Property	Description
<a href="#">154</a>	Create	Creates a new instance of the TEZCGI component
<a href="#">154</a>	Destroy	Removes the TEZCGI component from memory
<a href="#">156</a>	DoGet	Method to handle GET requests
<a href="#">156</a>	DoPost	Method to handle POST requests
<a href="#">156</a>	GetValue	Return the value of a request variable.
<a href="#">155</a>	PutLine	Send a line of output to the web-client
<a href="#">155</a>	Run	Run the CGI application.
<a href="#">155</a>	WriteContent	Writes the content type to standard output

### 11.5.3 Property overview

Page	Property	Access	Description
<a href="#">158</a>	Email	rw	Email of the server administrator
<a href="#">158</a>	Name	rw	Name of the server administrator
<a href="#">157</a>	Names	r	Indexed array with available variable names.
<a href="#">156</a>	Values	r	Variables passed to the CGI script
<a href="#">158</a>	VariableCount	r	Number of available variables.
<a href="#">157</a>	Variables	r	Indexed array with variables as name=value pairs.

### 11.5.4 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` (153) property.

See also: `TZECGI.Variables` (153), `TZECGI.Names` (153), `TZECGI.Values` (153)

### 11.5.5 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` ([154](#))

### 11.5.6 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` ([153](#)) property) and calls the `DoPost` ([156](#)) or `DoGet` ([156](#)) 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: `TEZCGI.Variables` ([153](#)), `TEZCGI.DoPost` ([156](#)), `TEZCGI.DoGet` ([156](#))

### 11.5.7 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` ([156](#)) or `DoGet` ([156](#)) methods.

See also: `TEZCGI.DoPost` ([156](#)), `TEZCGI.DoGet` ([156](#)), `TEZcgi.PutLine` ([155](#))

### 11.5.8 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` ([155](#)) 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` ([155](#))

### 11.5.9 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` ([156](#))

### 11.5.10 TEZcgi.DoPost

Synopsis: Method to handle POST requests

Declaration: `procedure DoPost; Virtual`

Visibility: public

Description: `DoPost` is called by the `Run` ([155](#)) 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` ([155](#)), `TEZcgi.DoGet` ([156](#))

### 11.5.11 TEZcgi.DoGet

Synopsis: Method to handle GET requests

Declaration: `procedure DoGet; Virtual`

Visibility: public

Description: `DoGet` is called by the `Run` ([155](#)) 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` ([155](#)), `TEZcgi.DoPost` ([156](#))

### 11.5.12 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\_HOST**DNS 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 \(157\)](#), [TEZCGI.Names \(157\)](#), [TEZCGI.GetValue \(156\)](#), [TEZcgi.VariableCount \(158\)](#)

### 11.5.13 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` ([158](#)). Any other value will result in an exception being raised.

See also: [TEZcgi.Variables \(157\)](#), [TEZcgi.Values \(156\)](#), [TEZcgi.GetValue \(156\)](#), [TEZcgi.VariableCount \(158\)](#)

### 11.5.14 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` (158). Any other value will result in an exception being raised.

See also: `TEZcgi.Names` (157), `TEZcgi.Values` (156), `TEZcgi.GetValue` (156), `TEZcgi.VariableCount` (158)

### 11.5.15 `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` (157) and `Variables` (157) properties.

See also: `TEZcgi.Names` (157), `TEZcgi.Variables` (157), `TEZcgi.Values` (156), `TEZcgi.GetValue` (156)

### 11.5.16 `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` (155) method.

See also: `TEZcgi.Run` (155), `TEZcgi.Email` (158)

### 11.5.17 `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` (155) method.

See also: `TEZcgi.Run` (155), `TEZcgi.Name` (158)

## Chapter 12

# Reference for unit 'gettext'

### 12.1 Used units

Table 12.1: Used units by unit 'gettext'

Name	Page
Classes	??
sysutils	??

### 12.2 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` ([161](#))) 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.

### 12.3 Constants, types and variables

#### 12.3.1 Constants

```
MOFileHeaderMagic = $950412de
```

This constant is found as the first integer in a `.mo`

#### 12.3.2 Types

```
PLongWordArray = ^TLongWordArray
```

Pointer to a `TLongWordArray` ([160](#)) array.

```
PMOStringTable = ^TMOStringTable
```

Pointer to a TMOStringTable (160) array.

```
PPCharArray = ^TPCharArray
```

Pointer to a TPCharArray (160) array.

```
TLongWordArray = Array[0..(1 shl 30) div SizeOf(LongWord)] of LongWord
```

TLongWordArray is an array used to define the PLongWordArray (159) 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 (160) record at the start of the file.

```
TMOStringTable = Array[0..(1 shl 30) div SizeOf(TMOStringInfo)] of TMOStringInfo
```

TMOStringTable is an array type containing TMOStringInfo (160) records. It should never be used directly, as it would occupy too much memory.

```
TPCharArray = Array[0..(1 shl 30) div SizeOf(PChar)] of PChar
```

TLongWordArray is an array used to define the PPCharArray (160) 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.

## 12.4 Procedures and functions

### 12.4.1 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.

## 12.4.2 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` (161) 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` (161), `TMOFile` (161)

## 12.4.3 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` (161), 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` (161), `TMOFile` (161)

## 12.5 EMOFileError

### 12.5.1 Description

`EMOFileError` is raised in case an `TMOFile` (161) instance is created with an invalid `.mo`.

## 12.6 TMOFile

### 12.6.1 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.

### 12.6.2 Method overview

Page	Property	Description
<a href="#">162</a>	Create	Create a new instance of the <code>TMOFile</code> class.
<a href="#">162</a>	Destroy	Removes the <code>TMOFile</code> instance from memory
<a href="#">162</a>	Translate	Translate a string

### 12.6.3 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` ([160](#)) structure, then an `EMOFileError` ([161](#)) exception is raised.

See also: `TMOFile.Destroy` ([162](#))

### 12.6.4 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` ([162](#))

### 12.6.5 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 13

# Reference for unit 'idea'

### 13.1 Used units

Table 13.1: Used units by unit 'idea'

Name	Page
Classes	??
sysutils	??

### 13.2 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 decryption stream which decrypts anything that is read from it.

### 13.3 Constants, types and variables

#### 13.3.1 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

### 13.3.2 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` (165) or `DeKeyIDEA` (164) 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.

## 13.4 Procedures and functions

### 13.4.1 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` (165), `DeKeyIdea` (164), `TIDEAEncryptStream` (166), `TIDEADecryptStream` (165)

### 13.4.2 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` ([165](#)), `CipherIdea` ([164](#))

### 13.4.3 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` ([164](#)), `CipherIdea` ([164](#))

## 13.5 EIDEAError

### 13.5.1 Description

`EIDEAError` is used to signal errors in the IDEA encryption decryption streams.

## 13.6 TIDEADeCryptStream

### 13.6.1 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` ([164](#)) 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` ([166](#)) encryption stream.

### 13.6.2 Method overview

Page	Property	Description
<a href="#">165</a>	Read	Reads data from the stream, decrypting it as needed
<a href="#">166</a>	Seek	Set position on the stream

### 13.6.3 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` (165), `TIDEADecryptStream.Seek` (166), `TIDEAEncryptStream` (166)

### 13.6.4 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` (165) exception is raised if the stream does not allow the requested seek operation.

**See also:** `TIDEADeCryptStream.Read` (165)

## 13.7 TIDEAEncryptStream

### 13.7.1 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` (164) 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` (165) decryption stream.

### 13.7.2 Method overview

Page	Property	Description
166	<code>Destroy</code>	Flush data buffers and free the stream instance.
167	<code>Flush</code>	Write remaining bytes from the stream
167	<code>Seek</code>	Set stream position
167	<code>Write</code>	Write bytes to the stream to be encrypted

### 13.7.3 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` (168)

### 13.7.4 `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` (165)

### 13.7.5 `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` (165) will be raised unless called with 0 and `soFromCurrent` as arguments.

See also: `TIDEAEncryptStream.Write` (167), `EIDEAError` (165)

### 13.7.6 `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` (167)

## 13.8 TIDEAStream

### 13.8.1 Description

Do not create instances of `TIDEAStream` directly. It implements no useful functionality: it serves as a common ancestor of the `TIDEAEncryptStream` ([166](#)) and `TIDEADecryptStream` ([165](#)), 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.

### 13.8.2 Method overview

Page	Property	Description
<a href="#">168</a>	Create	Creates a new instance of the <code>TIDEAStream</code> class

### 13.8.3 Property overview

Page	Property	Access	Description
<a href="#">168</a>	Key	r	Key used when encrypting/decrypting

### 13.8.4 TIDEAStream.Create

Synopsis: Creates a new instance of the `TIDEAStream` class

Declaration: constructor `Create(AKey: TIDEAKey; Dest: TStream)`

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` ([166](#)), `TIDEADecryptStream` ([165](#))

### 13.8.5 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` ([164](#))

## Chapter 14

# Reference for unit 'inicol'

### 14.1 Used units

Table 14.1: Used units by unit 'inicol'

Name	Page
Classes	??
Inifiles	<a href="#">169</a>
sysutils	??

### 14.2 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` ([179](#)) class for this.

### 14.3 Constants, types and variables

#### 14.3.1 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.

## 14.4 EIniCol

### 14.4.1 Description

EIniCol is used to report error conditions in the load and save methods of TIniCollection (170).

## 14.5 TIniCollection

### 14.5.1 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 (173).

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.

### 14.5.2 Method overview

Page	Property	Description
170	Load	Loads the collection from the default filename.
172	LoadFromFile	Load collection from file.
172	LoadFromIni	Load collection from a file in .ini file format.
171	Save	Save the collection to the default filename.
171	SaveToFile	Save collection to a file in .ini file format
171	SaveToIni	Save the collection to a TCustomIniFile descendent

### 14.5.3 Property overview

Page	Property	Access	Description
173	FileName	rw	Filename of the collection
173	GlobalSection	rw	Name of the global section
172	Prefix	r	Prefix used in global section
173	SectionPrefix	r	Prefix string for section names

### 14.5.4 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 (173) property. It calls the LoadFromFile (172) 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 (172), TIniCollection.LoadFromIni (172), TIniCollection.Save (171), TIniCollection.FileName (173)

### 14.5.5 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` (173) property, using `GlobalSection` (173) as the section. It calls the `SaveToFile` (171) 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` (171), `TIniCollection.SaveToIni` (171), `TIniCollection.Load` (170), `TIniCollection.FileName` (173)

### 14.5.6 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` (172) property for the key. After this it calls the `SaveToIni` (174) method of all `TIniCollectionItem` (173) 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` (173) property contains the value of `Section`, it will be used in the `Save` (173) method.

See also: `TIniCollectionItem.SaveToIni` (174)

### 14.5.7 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` (171) 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` (171), `TIniCollection.LoadFromFile` (172)

### 14.5.8 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` (174) 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` (171) method description.

See also: `TIniCollection.LoadFromFile` (172), `TIniCollectionItem.LoadFromIni` (174), `TIniCollection.SaveToIni` (171)

### 14.5.9 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` (172) using this instance and `Section` as the parameters.

See also: `TIniCollection.LoadFromIni` (172), `TIniCollection.Load` (170), `TIniCollection.SaveToIni` (171), `TIniCollection.SaveToFile` (171)

### 14.5.10 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` (173), `TIniCollection.GlobalSection` (173)

### 14.5.11 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` (175) property to return the exact section name. It can be empty.

See also: `TIniCollection.Section` (170), `TIniCollection.GlobalSection` (173)

### 14.5.12 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` (172) or `SaveToFile` (171) operation. It is used in the `Load` (170) or `Save` (171) calls.

See also: `TIniCollection.Save` (171), `TIniCollection.LoadFromFile` (172), `TIniCollection.SaveToFile` (171), `TIniCollection.Load` (170)

### 14.5.13 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` (172) or `SaveToIni` (171) calls. It's used in the `Load` (170) or `Save` (171) calls.

See also: `TIniCollection.Save` (171), `TIniCollection.LoadFromFile` (172), `TIniCollection.SaveToFile` (171), `TIniCollection.Load` (170)

## 14.6 TIniCollectionItem

### 14.6.1 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` (174) and `LoadFromIni` (174) methods should be overridden. They should implement the actual loading and saving. The loading and saving is always initiated by the methods in `TIniCollection` (170), `TIniCollection.LoadFromIni` (172) and `TIniCollection.SaveToIni` (171) respectively.

### 14.6.2 Method overview

Page	Property	Description
<a href="#">175</a>	LoadFromFile	Load item from a file
<a href="#">174</a>	LoadFromIni	Method called when the item must be loaded
<a href="#">174</a>	SaveToFile	Save item to a file
<a href="#">174</a>	SaveToIni	Method called when the item must be saved

### 14.6.3 Property overview

Page	Property	Access	Description
<a href="#">175</a>	SectionName	rw	Default section name

### 14.6.4 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` ([171](#)) 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` ([171](#)), `TIniCollectionItem.LoadFromIni` ([174](#)), `TIniCollectionItem.SaveToFile` ([174](#)), `TIniCollectionItem.LoadFromFile` ([175](#))

### 14.6.5 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` ([172](#)) 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` ([172](#)), `TIniCollectionItem.SaveToIni` ([174](#)), `TIniCollectionItem.LoadFromFile` ([175](#)), `TIniCollectionItem.SaveToFile` ([174](#))

### 14.6.6 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` (174) 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` (174), `TIniCollectionItem.LoadFromFile` (175)

### 14.6.7 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` (174) to load the item from the indicated file in .ini format from the section `Section`.

Errors: None.

See also: `TIniCollectionItem.SaveToFile` (174), `TIniCollectionItem.LoadFromIni` (174)

### 14.6.8 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` (174), `TIniCollectionItem.LoadFromIni` (174)

## 14.7 TNamedIniCollection

### 14.7.1 Description

`TNamedIniCollection` is the collection to go with the `TNamedIniCollectionItem` (177) item class. it provides some functions to look for items based on the `UserData` (176) or based on the `Name` (176).

### 14.7.2 Method overview

Page	Property	Description
176	<code>FindByName</code>	Return the item based on its name
177	<code>FindByUserData</code>	Return the item based on its <code>UserData</code>
176	<code>IndexOfName</code>	Search for an item, based on its name, and return its position
176	<code>IndexOfUserData</code>	Search for an item based on it's <code>UserData</code> property

### 14.7.3 Property overview

Page	Property	Access	Description
<a href="#">177</a>	NamedItems	rw	Indexed access to the <code>TNamedIniCollectionItem</code> items

### 14.7.4 `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` ([176](#)) 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` ([176](#)), `TNamedIniCollectionItem.UserData` ([177](#))

### 14.7.5 `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 insensitive). 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` ([176](#)), `TNamedIniCollectionItem.Name` ([178](#))

### 14.7.6 `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` ([176](#)) 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` ([176](#)), `TNamedIniCollection.FindByUserData` ([177](#))

### 14.7.7 TNamedIniCollection.FindByUserData

Synopsis: Return the item based on its `UserData`

Declaration: `function FindByUserData(UserData: TObject) : TNamedIniCollectionItem`

Visibility: `public`

Description: `FindByUserData` returns the collection item whose `UserData` (177) property value matches the `UserData` parameter. If no item is found, `Nil` is returned.

Errors: If no item is found, `Nil` is returned.

### 14.7.8 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` (177) items. The index is zero based.

See also: `TNamedIniCollectionItem` (177)

## 14.8 TNamedIniCollectionItem

### 14.8.1 Description

`TNamedIniCollectionItem` is a `TIniCollectionItem` (173) descent with a published name property. The name is used as the section name when saving the item to the ini file.

### 14.8.2 Property overview

Page	Property	Access	Description
<a href="#">178</a>	<code>Name</code>	<code>rw</code>	Name of the item
<a href="#">177</a>	<code>UserData</code>	<code>rw</code>	User-defined data

### 14.8.3 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`.

#### 14.8.4 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` ([177](#))

## Chapter 15

# Reference for unit 'IniFiles'

### 15.1 Used units

Table 15.1: Used units by unit 'IniFiles'

Name	Page
Classes	??
contnrs	<a href="#">67</a>
sysutils	??

### 15.2 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 ([179](#)) and is implemented in TIniFile ([191](#)) and TMemIniFile ([199](#)). The API presented by these components is Delphi compatible.

### 15.3 TCustomIniFile

#### 15.3.1 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 ([191](#)) or TMemIniFile ([199](#)).

Since TCustomIniFile is an abstract class, it should never be created directly. Instead, one of the TIniFile or TMemIniFile classes should be created.

### 15.3.2 Method overview

Page	Property	Description
<a href="#">180</a>	Create	Instantiate a new instance of TCustomIniFile.
<a href="#">187</a>	DeleteKey	Delete a key from a section
<a href="#">181</a>	Destroy	Remove the TCustomIniFile instance from memory
<a href="#">187</a>	EraseSection	Clear a section
<a href="#">184</a>	ReadBinaryStream	Read binary data
<a href="#">182</a>	ReadBool	
<a href="#">183</a>	ReadDate	Read a date value
<a href="#">183</a>	ReadDateTime	Read a Date/Time value
<a href="#">184</a>	ReadFloat	Read a floating point value
<a href="#">182</a>	ReadInteger	Read an integer value from the file
<a href="#">186</a>	ReadSection	Read the key names in a section
<a href="#">187</a>	ReadSections	Read the list of sections
<a href="#">187</a>	ReadSectionValues	Read names and values of a section
<a href="#">181</a>	ReadString	Read a string valued key
<a href="#">184</a>	ReadTime	Read a time value
<a href="#">181</a>	SectionExists	Check if a section exists.
<a href="#">188</a>	UpdateFile	Update the file on disk
<a href="#">188</a>	ValueExists	Check if a value exists
<a href="#">186</a>	WriteBinaryStream	Write binary data
<a href="#">183</a>	WriteBool	Write boolean value
<a href="#">185</a>	WriteDate	Write date value
<a href="#">185</a>	WriteDateTime	Write date/time value
<a href="#">185</a>	WriteFloat	Write a floating-point value
<a href="#">182</a>	WriteInteger	Write an integer value
<a href="#">182</a>	WriteString	Write a string value
<a href="#">186</a>	WriteTime	Write time value

### 15.3.3 Property overview

Page	Property	Access	Description
<a href="#">189</a>	CaseSensitive	rw	Are key and section names case sensitive
<a href="#">189</a>	EscapeLineFeeds	r	Should linefeeds be escaped ?
<a href="#">188</a>	FileName	r	Name of the .ini file
<a href="#">189</a>	StripQuotes	rw	Should quotes be stripped from string values

### 15.3.4 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` ([181](#))

### 15.3.5 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` ([179](#))

### 15.3.6 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` ([188](#))

### 15.3.7 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` ([182](#)), `TCustomIniFile.ReadInteger` ([182](#)), `TCustomIniFile.ReadBool` ([182](#)), `TCustomIniFile.ReadDate` ([183](#)), `TCustomIniFile.ReadDateTime` ([183](#)), `TCustomIniFile.ReadTime` ([184](#)), `TCustomIniFile.ReadFloat` ([184](#)), `TCustomIniFile.ReadBinaryStream` ([184](#))

### 15.3.8 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` (181), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.9 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` (182), `TCustomIniFile.ReadString` (181), `TCustomIniFile.ReadBool` (182), `TCustomIniFile.ReadDate` (183), `TCustomIniFile.ReadDateTime` (183), `TCustomIniFile.ReadTime` (184), `TCustomIniFile.ReadFloat` (184), `TCustomIniFile.ReadBinaryStream` (184)

### 15.3.10 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` (182), `TCustomIniFile.WriteString` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.11 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` (183), `TCustomIniFile.ReadInteger` (182), `TCustomIniFile.ReadString` (181), `TCustomIniFile.ReadDate` (183), `TCustomIniFile.ReadDateTime` (183), `TCustomIniFile.ReadTime` (184), `TCustomIniFile.ReadFloat` (184), `TCustomIniFile.ReadBinaryStream` (184)

### 15.3.12 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` (182), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteString` (182), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.13 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` (185), `TCustomIniFile.ReadInteger` (182), `TCustomIniFile.ReadBool` (182), `TCustomIniFile.ReadString` (181), `TCustomIniFile.ReadDateTime` (183), `TCustomIniFile.ReadTime` (184), `TCustomIniFile.ReadFloat` (184), `TCustomIniFile.ReadBinaryStream` (184)

### 15.3.14 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` (185), `TCustomIniFile.ReadInteger` (182), `TCustomIniFile.ReadBool` (182), `TCustomIniFile.ReadDate` (183), `TCustomIniFile.ReadString` (181), `TCustomIniFile.ReadTime` (184), `TCustomIniFile.ReadFloat` (184), `TCustomIniFile.ReadBinaryStream` (184)

### 15.3.15 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` (185), `TCustomIniFile.ReadInteger` (182), `TCustomIniFile.ReadBool` (182), `TCustomIniFile.ReadDate` (183), `TCustomIniFile.ReadDateTime` (183), `TCustomIniFile.ReadTime` (184), `TCustomIniFile.ReadString` (181), `TCustomIniFile.ReadBinaryStream` (184)

### 15.3.16 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` (186), `TCustomIniFile.ReadInteger` (182), `TCustomIniFile.ReadBool` (182), `TCustomIniFile.ReadDate` (183), `TCustomIniFile.ReadDateTime` (183), `TCustomIniFile.ReadString` (181), `TCustomIniFile.ReadFloat` (184), `TCustomIniFile.ReadBinaryStream` (184)

### 15.3.17 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` (186), `TCustomIniFile.ReadInteger` (182), `TCustomIniFile.ReadBool` (182), `TCustomIniFile.ReadDate` (183), `TCustomIniFile.ReadDateTime` (183), `TCustomIniFile.ReadTime` (184), `TCustomIniFile.ReadFloat` (184), `TCustomIniFile.ReadString` (181)

### 15.3.18 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` (183), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteString` (182), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.19 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` (183), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteString` (182), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.20 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` (184), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteString` (182), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.21 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` (184), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteString` (182), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteBinaryStream` (186)

### 15.3.22 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` (184), `TCustomIniFile.WriteInteger` (182), `TCustomIniFile.WriteBool` (183), `TCustomIniFile.WriteDate` (185), `TCustomIniFile.WriteDateTime` (185), `TCustomIniFile.WriteTime` (186), `TCustomIniFile.WriteFloat` (185), `TCustomIniFile.WriteString` (182)

### 15.3.23 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` (187) method should be used.

See also: `TCustomIniFile.ReadSections` ([187](#)), `TCustomIniFile.SectionExists` ([181](#)), `TCustomIniFile.ReadSectionValues` ([187](#))

### 15.3.24 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` ([181](#)), `TCustomIniFile.ReadSectionValues` ([187](#)), `TCustomIniFile.ReadSection` ([186](#))

### 15.3.25 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` ([186](#)) can be used.

See also: `TCustomIniFile.SectionExists` ([181](#)), `TCustomIniFile.ReadSections` ([187](#)), `TCustomIniFile.ReadSection` ([186](#))

### 15.3.26 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` ([181](#)), `TCustomIniFile.ReadSections` ([187](#)), `TCustomIniFile.DeleteKey` ([187](#))

### 15.3.27 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` ([187](#))

### 15.3.28 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 immediately 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` ([194](#))

### 15.3.29 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` ([181](#))

### 15.3.30 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` ([180](#))

### 15.3.31 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 (180), as the value of this property must be specified when the TCustomIniFile instance is created.

By default, EscapeLineFeeds is False.

See also: TCustomIniFile.Create (180), TCustomIniFile.CaseSensitive (189)

### 15.3.32 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 (189)

### 15.3.33 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).

## 15.4 THashedStringList

### 15.4.1 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 (190) and IndexOfName (190) functions make use of these hash values to quicklier locate a value.

### 15.4.2 Method overview

Page	Property	Description
<a href="#">190</a>	Create	Instantiates a new instance of THashedStringList
<a href="#">190</a>	Destroy	Clean up instance
<a href="#">190</a>	IndexOf	Returns the index of a string in the list of strings
<a href="#">190</a>	IndexOfName	Return the index of a name in the list of name=value pairs

### 15.4.3 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 ([190](#))

### 15.4.4 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 ([190](#))

### 15.4.5 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 ([179](#)) method and uses the hash values to look for the location of S.

See also: TStringList.IndexOf ([179](#)), THashedStringList.IndexOfName ([190](#))

### 15.4.6 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 TStringList.IndexOfName ([179](#)) method and uses the hash values of the names to look for the location of Name.

See also: TStringList.IndexOfName ([179](#)), THashedStringList.IndexOf ([190](#))

## 15.5 TIniFile

### 15.5.1 Description

`TIniFile` is an implementation of `TCustomIniFile` (179) which does the same as `TMemIniFile` (199), namely it reads the whole file into memory. Unlike `TMemIniFile` it does not cache updates in memory, but immediately writes any changes to disk.

`TIniFile` introduces no new methods, it just implements the abstract methods introduced in `TCustomIniFile`

### 15.5.2 Method overview

Page	Property	Description
<a href="#">191</a>	Create	Create a new instance of <code>TIniFile</code>
<a href="#">193</a>	DeleteKey	Delete key
<a href="#">191</a>	Destroy	Remove the <code>TIniFile</code> instance from memory
<a href="#">193</a>	EraseSection	
<a href="#">192</a>	ReadSection	Read the key names in a section
<a href="#">192</a>	ReadSectionRaw	Read raw section
<a href="#">193</a>	ReadSections	Read section names
<a href="#">193</a>	ReadSectionValues	
<a href="#">192</a>	ReadString	Read a string
<a href="#">194</a>	UpdateFile	Update the file on disk
<a href="#">192</a>	WriteString	Write string to file

### 15.5.3 Property overview

Page	Property	Access	Description
<a href="#">194</a>	CacheUpdates	rw	Should changes be kept in memory
<a href="#">194</a>	Stream	r	Stream from which ini file was read

### 15.5.4 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` (179), `TMemIniFile` (199)

### 15.5.5 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` ([188](#)), `TIniFile.CacheUpdates` ([194](#))

### 15.5.6 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` ([181](#)) abstract method by looking at the in-memory copy of the ini file and returning the string found there.

**See also:** `TCustomIniFile.ReadString` ([181](#))

### 15.5.7 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` ([182](#)) abstract method by writing the string to the in-memory copy of the ini file. If `CacheUpdates` ([194](#)) 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.

### 15.5.8 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` ([186](#))

**See also:** `TCustomIniFile.ReadSection` ([186](#)), `TIniFile.ReadSectionRaw` ([192](#))

### 15.5.9 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` ([192](#)), `TCustomIniFile.ReadSection` ([186](#))

### 15.5.10 TIniFile.ReadSections

**Synopsis:** Read section names

**Declaration:** `procedure ReadSections(Strings: TStrings); Override`

**Visibility:** public

**Description:** `ReadSections` is the implementation of `TCustomIniFile.ReadSections` ([187](#)). It operates on the in-memory copy of the inifile, and places all section names in `Strings`.

See also: `TIniFile.ReadSection` ([192](#)), `TCustomIniFile.ReadSections` ([187](#)), `TIniFile.ReadSectionValues` ([193](#))

### 15.5.11 TIniFile.ReadSectionValues

**Synopsis:**

**Declaration:** `procedure ReadSectionValues(const Section: String; Strings: TStrings)  
; Override`

**Visibility:** public

**Description:** `ReadSectionValues` is the implementation of `TCustomIniFile.ReadSectionValues` ([187](#)). 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` ([192](#)), `TCustomIniFile.ReadSectionValues` ([187](#)), `TIniFile.ReadSections` ([193](#))

### 15.5.12 TIniFile.EraseSection

**Synopsis:**

**Declaration:** `procedure EraseSection(const Section: String); Override`

**Visibility:** public

**Description:** `EraseSection` deletes the section `Section` from memory, if `CacheUpdates` ([194](#)) is `False`, then the file is immediatly updated on disk. This method is the implementation of the abstract `TCustomIniFile.EraseSection` ([187](#)) method.

See also: `TCustomIniFile.EraseSection` ([187](#)), `TIniFile.ReadSection` ([192](#)), `TIniFile.ReadSections` ([193](#))

### 15.5.13 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` (194) is `False`, then the file is immediatly updated on disk.

See also: `TIniFile.CacheUpdates` (194)

### 15.5.14 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` (194)

### 15.5.15 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` (194) method will use this stream to write changes to.

See also: `TIniFile.Create` (191), `TIniFile.UpdateFile` (194)

### 15.5.16 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` (194). 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` (194)

## 15.6 TIniFileKey

### 15.6.1 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 (191) class.

### 15.6.2 Method overview

Page	Property	Description
<a href="#">195</a>	Create	Create a new instance of TIniFileKey

### 15.6.3 Property overview

Page	Property	Access	Description
<a href="#">195</a>	Ident	rw	Key name
<a href="#">195</a>	Value	rw	Key value

### 15.6.4 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 ([195](#)) with AIdent and Value ([195](#)) with AValue.

See also: TIniFileKey.Ident ([195](#)), TIniFileKey.Value ([195](#))

### 15.6.5 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 ([195](#))

### 15.6.6 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 ([195](#))

## 15.7 TIniFileKeyList

### 15.7.1 Description

TIniFileKeyList maintains a list of TIniFileKey (195) instances on behalf of the TIniFileSection (197) class. It stores the keys of one section of the .ini files.

### 15.7.2 Method overview

Page	Property	Description
<a href="#">196</a>	Clear	Clear the list
<a href="#">196</a>	Destroy	Free the instance

### 15.7.3 Property overview

Page	Property	Access	Description
<a href="#">196</a>	Items	r	Indexed access to TIniFileKey items in the list

### 15.7.4 TIniFileKeyList.Destroy

Synopsis: Free the instance

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: Destroy clears up the list using Clear (196) and then calls the inherited destroy.

See also: TIniFileKeyList.Clear (196)

### 15.7.5 TIniFileKeyList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear; Override`

Visibility: `public`

Description: Clear removes all TIniFileKey (195) instances from the list, and frees the instances.

See also: TIniFileKey (195)

### 15.7.6 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 (195) items in the list. The index is zero-based and runs from 0 to Count-1.

See also: TIniFileKey (195)

## 15.8 TIniFileSection

### 15.8.1 Description

TIniFileSection is a class which represents a section in the .ini, and is used internally by the TIniFile (191) class (one instance of TIniFileSection is created for each section in the file by the TIniFileSectionList (198) list). The name of the section is stored in the Name (198) property, and the key/value pairs in this section are available in the KeyList (198) property.

### 15.8.2 Method overview

Page	Property	Description
197	Create	Create a new section object
197	Destroy	Free the section object from memory
197	Empty	Is the section empty

### 15.8.3 Property overview

Page	Property	Access	Description
198	KeyList	r	List of key/value pairs in this section
198	Name	r	Name of the section

### 15.8.4 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.

### 15.8.5 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 (196) instance to keep all the key/value pairs for this section.

See also: TIniFileKeyList (196)

### 15.8.6 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` ([197](#)), `TIniFileKeyList` ([196](#))

### 15.8.7 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` ([198](#))

### 15.8.8 TIniFileSection.KeyList

**Synopsis:** List of key/value pairs in this section

**Declaration:** `Property KeyList : TIniFileKeyList`

**Visibility:** `public`

**Access:** `Read`

**Description:** `KeyList` is the `TIniFileKeyList` ([196](#)) instance that is used by the `TIniFileSection` to keep the key/value pairs of the section.

See also: `TIniFileSection.Name` ([198](#)), `TIniFileKeyList` ([196](#))

## 15.9 TIniFileSectionList

### 15.9.1 Description

`TIniFileSectionList` maintains a list of `TIniFileSection` ([197](#)) instances, one for each section in an .ini file. `TIniFileSectionList` is used internally by the `TIniFile` ([191](#)) class to represent the sections in the file.

### 15.9.2 Method overview

Page	Property	Description
<a href="#">199</a>	<code>Clear</code>	Clear the list
<a href="#">199</a>	<code>Destroy</code>	Free the object from memory

### 15.9.3 Property overview

Page	Property	Access	Description
<a href="#">199</a>	<code>Items</code>	<code>r</code>	Indexed access to all the section objects in the list

### 15.9.4 TIniFileSectionList.Destroy

Synopsis: Free the object from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` calls `Clear` (199) to clear the section list and then calls the inherited `Destroy`

See also: `TIniFileSectionList.Clear` (199)

### 15.9.5 TIniFileSectionList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear; Override`

Visibility: `public`

Description: `Clear` removes all `TIniFileSection` (197) items from the list, and frees the items it removes from the list.

See also: `TIniFileSection` (197), `TIniFileSectionList.Items` (199)

### 15.9.6 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` (197), `TIniFileSectionList.Clear` (199)

## 15.10 TMemIniFile

### 15.10.1 Description

`TMemIniFile` is a simple descendent of `TIniFile` (191) 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.

### 15.10.2 Method overview

Page	Property	Description
<a href="#">200</a>	<code>Clear</code>	Clear the data
<a href="#">200</a>	<code>Create</code>	Create a new instance of <code>TMemIniFile</code>
<a href="#">200</a>	<code>GetStrings</code>	Get contents of ini file as stringlist
<a href="#">200</a>	<code>Rename</code>	Rename the ini file
<a href="#">201</a>	<code>SetStrings</code>	Set data from a stringlist

### 15.10.3 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` (191), and sets the `CacheUpdates` (194) to `True` so updates will be kept in memory till they are explicitly written to disk.

See also: `TIniFile.Create` (191), `TIniFile.CacheUpdates` (194)

### 15.10.4 TMemIniFile.Clear

Synopsis: Clear the data

Declaration: `procedure Clear`

Visibility: public

Description: `Clear` removes all sections and key/value pairs from memory. If `CacheUpdates` (194) is set to `False` then the file on disk will immediatly be emptied.

See also: `TMemIniFile.SetStrings` (201), `TMemIniFile.GetStrings` (200)

### 15.10.5 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` (201), which can be used to copy data between 2 in-memory ini files.

See also: `TMemIniFile.SetStrings` (201), `TMemIniFile.Clear` (200)

### 15.10.6 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` (194)

### 15.10.7 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` (200). The two calls combined can be used to copy data between 2 in-memory ini files.

See also: `TMemIniFile.GetStrings` (200), `TMemIniFile.Clear` (200)

## Chapter 16

# Reference for unit 'iostream'

### 16.1 Used units

Table 16.1: Used units by unit 'iostream'

Name	Page
Classes	??

### 16.2 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`).

### 16.3 Constants, types and variables

#### 16.3.1 Types

```
TIOSType = (iosInput, iosOutPut, iosError)
```

Table 16.2: Enumeration values for type `TIOSType`

Value	Explanation
<code>iosError</code>	The stream can be used to write to standard diagnostic output
<code>iosInput</code>	The stream can be used to read from standard input
<code>iosOutPut</code>	The stream can be used to write to standard output

`TIOSType` is passed to the `Create` (203) constructor of `TIOStream` (203), it determines what kind of stream is created.

## 16.4 EIOStreamError

### 16.4.1 Description

Error thrown in case of an invalid operation on a TIOStream ([203](#)).

## 16.5 TIOStream

### 16.5.1 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 ([202](#)) 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 ([204](#)) behaviour based on this position.

### 16.5.2 Method overview

Page	Property	Description
<a href="#">203</a>	Create	Construct a new instance of TIOStream ( <a href="#">203</a> )
<a href="#">203</a>	Read	Read data from the stream.
<a href="#">204</a>	Seek	Set the stream position
<a href="#">204</a>	SetSize	Set the size of the stream
<a href="#">204</a>	Write	Write data to the stream

### 16.5.3 TIOStream.Create

Synopsis: Construct a new instance of TIOStream ([203](#))

Declaration: `constructor Create(aIOSType: TIOSType)`

Visibility: public

Description: Create creates a new instance of TIOStream ([203](#)), 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 ([203](#)), TIOStream.Write ([204](#))

### 16.5.4 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 ([203](#)) 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 ([202](#)), TIOStream.Write ([204](#))

### 16.5.5 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` (203) 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: `TIOStreamType` (202), `TIOStream.Read` (203)

### 16.5.6 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` (203)

### 16.5.7 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` (203) exception is raised if the stream does not allow the requested seek operation.

See also: `EIOStreamError` (203)

# Chapter 17

## Reference for unit 'Pipes'

### 17.1 Used units

Table 17.1: Used units by unit 'Pipes'

Name	Page
Classes	??
sysutils	??

### 17.2 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.

### 17.3 Constants, types and variables

#### 17.3.1 Constants

`ENoSeekMsg = 'Cannot seek on pipes'`

Constant used in `EPipeSeek` (206) exception.

`EPipeMsg = 'Failed to create pipe.'`

Constant used in `EPipeCreation` (206) exception.

### 17.4 Procedures and functions

#### 17.4.1 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` (206)

## 17.4.2 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` (205), and if that call is successful, a pair of streams is created: `InPipe` and `OutPipe`.

**Errors:** If no pipe handles could be created, an `EPipeCreation` (206) exception is raised.

**See also:** `CreatePipeHandles` (205), `TInputPipeStream` (206), `TOutputPipeStream` (208)

## 17.5 EPipeCreation

### 17.5.1 Description

Exception raised when an error occurred during the creation of a pipe pair.

## 17.6 EPipeError

### 17.6.1 Description

Exception raised when an invalid operation is performed on a pipe stream.

## 17.7 EPipeSeek

### 17.7.1 Description

Exception raised when an invalid seek operation is attempted on a pipe.

## 17.8 TInputPipeStream

### 17.8.1 Description

`TInputPipeStream` is created by the `CreatePipeStreams` (206) 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.

### 17.8.2 Method overview

Page	Property	Description
<a href="#">208</a>	Read	Read data from the stream to a buffer.
<a href="#">207</a>	Seek	Set the current position of the stream
<a href="#">207</a>	Write	Write data to the stream.

### 17.8.3 Property overview

Page	Property	Access	Description
<a href="#">208</a>	NumBytesAvailable	r	Number of bytes available for reading.

### 17.8.4 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` ([205](#)) exception is raised when this function is called.

See also: `TInputPipeStream.Read` ([208](#)), `TInputPipeStream.Seek` ([207](#))

### 17.8.5 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` ([206](#)) exception is raised if the stream does not allow the requested seek operation.

See also: `EPipeSeek` ([206](#)), `#rtl.classes.tstream.seek` (??)

### 17.8.6 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` (207), `TInputPipeStream.Seek` (207)

### 17.8.7 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` (208)

## 17.9 TOutputPipeStream

### 17.9.1 Description

`TOutputPipeStream` is created by the `CreatePipeStreams` (206) call to represent the writing end of a pipe. It is a `TStream` (??) descendent which does not allow reading.

### 17.9.2 Method overview

Page	Property	Description
<a href="#">209</a>	<code>Read</code>	Read data from the stream.
<a href="#">208</a>	<code>Seek</code>	Sets the position in the stream

### 17.9.3 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` (206) exception is raised if this method is called.

### 17.9.4 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` (205) exception is raised when this function is called.

See also: `TOutputPipeStream.Seek` (208)

## Chapter 18

# Reference for unit 'pooledmm'

### 18.1 Used units

Table 18.1: Used units by unit 'pooledmm'

Name	Page
Classes	??

### 18.2 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.

### 18.3 Constants, types and variables

#### 18.3.1 Types

`PPooledMemManagerItem` = `^TPooledMemManagerItem`

`PPooledMemManagerItem` is a pointer type, pointing to a `TPooledMemManagerItem` (211) item, used in a linked list.

`TEnumItemsMethod` = `procedure(Item: Pointer) of object`

`TEnumItemsMethod` is a prototype for the callback used in the `TNonFreePooledMemManager.EnumerateItems` (212) call. The parameter `Item` will be set to each of the pointers in the item list of `TNonFreePooledMemManager` (211).

```
TPooledMemManagerItem = record
  Next : PPooledMemManagerItem;
end
```

`TPooledMemManagerItem` is used internally by the `TPooledMemManager` (213) class to maintain the free list block. It simply points to the next free block.

## 18.4 TNonFreePooledMemManager

### 18.4.1 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 an exponential way, i.e. each time a new block of memory must be allocated, its size is the double of the last block. The first block will contain 8 items.

### 18.4.2 Method overview

Page	Property	Description
<a href="#">211</a>	<code>Clear</code>	Clears the memory
<a href="#">211</a>	<code>Create</code>	Creates a new instance of <code>TNonFreePooledMemManager</code>
<a href="#">212</a>	<code>Destroy</code>	Removes the <code>TNonFreePooledMemManager</code> instance from memory
<a href="#">212</a>	<code>EnumerateItems</code>	Enumerate all items in the list
<a href="#">212</a>	<code>NewItem</code>	Return a pointer to a new memory block

### 18.4.3 Property overview

Page	Property	Access	Description
<a href="#">212</a>	<code>ItemSize</code>	<code>r</code>	Size of an item in the list

### 18.4.4 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` (212) is valid after a call to `Clear`

See also: `TNonFreePooledMemManager.NewItem` (212)

### 18.4.5 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` (212)

### 18.4.6 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` ([211](#)), `TNonFreePooledMemManager.Clear` ([211](#))

### 18.4.7 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` ([212](#)). 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` ([211](#))

### 18.4.8 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)

### 18.4.9 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` ([211](#))

## 18.5 TPooledMemManager

### 18.5.1 Description

`TPooledMemManager` is a class which maintains a linked list of blocks, represented by the `TPooledMemManagerItem` (211) record. It should not be used directly, but should be descended from and the descendent should implement the actual memory manager.

### 18.5.2 Method overview

Page	Property	Description
<a href="#">213</a>	<code>Clear</code>	Clears the list
<a href="#">213</a>	<code>Create</code>	Creates a new instance of the <code>TPooledMemManager</code> class
<a href="#">213</a>	<code>Destroy</code>	Removes an instance of <code>TPooledMemManager</code> class from memory

### 18.5.3 Property overview

Page	Property	Access	Description
<a href="#">215</a>	<code>AllocatedCount</code>	r	Total number of allocated items in the list
<a href="#">214</a>	<code>Count</code>	r	Number of items in the list
<a href="#">215</a>	<code>FreeCount</code>	r	Number of free items in the list
<a href="#">215</a>	<code>FreedCount</code>	r	Total number of freed items in the list.
<a href="#">214</a>	<code>MaximumFreeCountRatio</code>	rw	Maximum ratio of free items over total items
<a href="#">214</a>	<code>MinimumFreeCount</code>	rw	Minimum count of free items in the list

### 18.5.4 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` ([215](#))

### 18.5.5 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` ([213](#))

### 18.5.6 TPooledMemManager.Destroy

Synopsis: Removes an instance of `TPooledMemManager` class from memory

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` calls `Clear` (213) and then calls the inherited `destroy`.

`Destroy` should never be called directly. Instead `Free` should be used, or `FreeAndNil`

See also: `TPooledMemManager.Create` (213)

### 18.5.7 `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` (214)

### 18.5.8 `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` (214)

### 18.5.9 `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` (215), `TPooledMemManager.AllocatedCount` (215), `TPooledMemManager.FreedCount` (215)

### 18.5.10 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` (214), `TPooledMemManager.AllocatedCount` (215), `TPooledMemManager.FreedCount` (215)

### 18.5.11 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` (214), `TPooledMemManager.FreeCount` (215), `TPooledMemManager.FreedCount` (215)

### 18.5.12 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` (214), `TPooledMemManager.FreeCount` (215), `TPooledMemManager.AllocatedCount` (215)

## Chapter 19

# Reference for unit 'process'

### 19.1 Used units

Table 19.1: Used units by unit 'process'

Name	Page
Classes	??
Pipes	<a href="#">205</a>
sysutils	??

### 19.2 Overview

The `Process` unit contains the code for the `TProcess` ([218](#)) 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.

### 19.3 Constants, types and variables

#### 19.3.1 Types

```
TProcessOption = (poRunSuspended, poWaitOnExit, poUsePipes,  
                  poStderrToOutPut, poNoConsole, poNewConsole,  
                  poDefaultErrorMode, poNewProcessGroup, poDebugProcess,  
                  poDebugOnlyThisProcess)
```

When a new process is started using `TProcess.Execute` ([220](#)), 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 19.2: 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 (216).

```
TProcessPriority = (ppHigh, ppIdle, ppNormal, ppRealTime)
```

Table 19.3: 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 19.4: 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 19.5: Enumeration values for type TStartupOption

Value	Explanation
suoUseCountChars	Use the console character width as specified in TProcess (218).
suoUseFillAttribute	Use the console fill attribute as specified in TProcess (218).
suoUsePosition	Use the window sizes as specified in TProcess (218).
suoUseShowWindow	Use the Show Window options specified in TShowWindowOption (217)
suoUseSize	Use the window sizes as specified in TProcess (218)

```
TStartupOptions= Set of (suoUseCountChars,suoUseFillAttribute,
                        suoUsePosition,suoUseShowWindow,suoUseSize)
```

Set of TStartUpOption (217).

## 19.4 EProcess

### 19.4.1 Description

Exception raised when an error occurs in a TProcess routine.

## 19.5 TProcess

### 19.5.1 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 (226) property to the full pathname of the program that should be executed, and call Execute (220). To determine whether the process is still running (i.e. has not stopped executing), the Running (230) property can be checked.

More advanced techniques can be used with the Options (228) settings.

### 19.5.2 Method overview

Page	Property	Description
221	CloseInput	Close the input stream of the process
221	CloseOutput	Close the output stream of the process
221	CloseStderr	Close the error stream of the process
220	Create	Create a new instance of the <code>TProcess</code> class.
220	Destroy	Destroy this instance of <code>TProcess</code>
220	Execute	Execute the program with the given options
221	Resume	Resume execution of a suspended process
222	Suspend	Suspend a running process
222	Terminate	Terminate a running process
222	WaitOnExit	Wait for the program to stop executing.

### 19.5.3 Property overview

Page	Property	Access	Description
226	Active	rw	Start or stop the process.
226	ApplicationName	rw	Name of the application to start
226	CommandLine	rw	Command-line to execute
227	ConsoleTitle	rw	Title of the console window
227	CurrentDirectory	rw	Working directory of the process.
227	Desktop	rw	Desktop on which to start the process.
228	Environment	rw	Environment variables for the new process
225	ExitStatus	r	Exit status of the process.
233	FillAttribute	rw	Color attributes of the characters in the console window (Windows only)
223	Handle	r	Handle of the process
226	InheritHandles	rw	Should the created process inherit the open handles of the current process.
224	Input	r	Stream connected to standard input of the process.
228	Options	rw	Options to be used when starting the process.
225	Output	r	Stream connected to standard output of the process.
229	Priority	rw	Priority at which the process is running.
223	ProcessHandle	r	Alias for Handle (223)
224	ProcessID	r	ID of the process.
230	Running	r	Determines wheter the process is still running.
230	ShowWindow	rw	Determines how the process main window is shown (Windows only)
229	StartupOptions	rw	Additional (Windows) startup options
225	Stderr	r	Stream connected to standard diagnostic output of the process.
223	ThreadHandle	r	Main process thread handle
224	ThreadID	r	ID of the main process thread
231	WindowColumns	rw	Number of columns in console window (windows only)
231	WindowHeight	rw	Height of the process main window
231	WindowLeft	rw	X-coordinate of the initial window (Windows only)
223	WindowRect	rw	Positions for the main program window.
232	WindowRows	rw	Number of rows in console window (Windows only)
232	WindowTop	rw	Y-coordinate of the initial window (Windows only)
232	WindowWidth	rw	Height of the process main window (Windows only)

### 19.5.4 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.

### 19.5.5 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` (220)

### 19.5.6 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` (226), applying as much as of the specified options as supported on the current platform.

If the `poWaitOnExit` option is specified in `Options` (228), then the call will only return when the program has finished executing (or if an error occurred). If this option is not given, the call returns immediately, but the `WaitOnExit` (222) call can be used to wait for it to close, or the `Running` (230) call can be used to check whether it is still running.

The `TProcess.Terminate` (222) call can be used to terminate the program if it is still running, or the `Suspend` (222) call can be used to temporarily stop the program's execution.

The `ExitStatus` (225) function can be used to check the program's exit status, after it has stopped executing.

Errors: On error a `EProcess` (218) exception is raised.

See also: `TProcess.Running` (230), `TProcess.WaitOnExit` (222), `TProcess.Terminate` (222), `TProcess.Suspend` (222), `TProcess.Resume` (221), `TProcess.ExitStatus` (225)

### 19.5.7 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` (224), `TProcess.StdErr` (225), `TProcess.Output` (225), `TProcess.CloseOutput` (221), `TProcess.CloseStdErr` (221)

### 19.5.8 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` (225), `TProcess.Input` (224), `TProcess.StdErr` (225), `TProcess.CloseInput` (221), `TProcess.CloseStdErr` (221)

### 19.5.9 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` (225), `TProcess.Input` (224), `TProcess.StdErr` (225), `TProcess.CloseInput` (221), `TProcess.CloseStdErr` (221)

### 19.5.10 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 its execution. It should be called in particular when the `poRunSuspended` flag is set in `Options` (228).

Errors: None.

See also: `TProcess.Suspend` (222), `TProcess.Options` (228), `TProcess.Execute` (220), `TProcess.Terminate` (222)

### 19.5.11 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` (221) call.

`Suspend` is fundamentally different from `TProcess.Terminate` (222) which actually stops the process.

Errors: On error, a nonzero result is returned.

See also: `TProcess.Options` (228), `TProcess.Resume` (221), `TProcess.Terminate` (222), `TProcess.Execute` (220)

### 19.5.12 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` (225), `TProcess.Suspend` (222), `TProcess.Execute` (220), `TProcess.WaitOnExit` (222)

### 19.5.13 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 successful, 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` (225), `TProcess.Terminate` (222), `TProcess.Running` (230)

### 19.5.14 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

### 19.5.15 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` (220) has been called. It is not reset after the process stopped.

See also: `TProcess.ThreadHandle` (223), `TProcess.ProcessID` (224), `TProcess.ThreadID` (224)

### 19.5.16 TProcess.ProcessHandle

Synopsis: Alias for `Handle` (223)

Declaration: `Property ProcessHandle : THandle`

Visibility: `public`

Access: `Read`

Description: `ProcessHandle` equals `Handle` (223) and is provided for completeness only.

See also: `TProcess.Handle` (223), `TProcess.ThreadHandle` (223), `TProcess.ProcessID` (224), `TProcess.ThreadID` (224)

### 19.5.17 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` (220) has been called. It is not reset after the process stopped.

See also: `TProcess.Handle` (223), `TProcess.ProcessID` (224), `TProcess.ThreadID` (224)

### 19.5.18 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` (220) has been called. It is not reset after the process stopped.

See also: `TProcess.Handle` (223), `TProcess.ThreadHandle` (223), `TProcess.ThreadID` (224)

### 19.5.19 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 process 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` (220) has been called. It is not reset after the process stopped.

See also: `TProcess.ProcessID` (224), `TProcess.Handle` (223), `TProcess.ThreadHandle` (223)

### 19.5.20 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` (228).

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` (225), `TProcess.StdErr` (225), `TProcess.Options` (228), `TProcessOption` (216)

### 19.5.21 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` (228).

The `Output` stream also contains any data written to standard diagnostic output (`stderr`) when the `poStdErrToOutPut` flag is used in `Options` (228).

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` (224), `TProcess.StdErr` (225), `TProcess.Options` (228), `TProcessOption` (216)

### 19.5.22 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` (228).

The `Output` stream equals the `Output` (225) when the `poStdErrToOutPut` flag is used in `Options` (228).

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` (224), `TProcess.Output` (225), `TProcess.Options` (228), `TProcessOption` (216)

### 19.5.23 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` (230), `TProcess.Terminate` (222)

### 19.5.24 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` (224), `TProcess.Output` (225), `TProcess.StdErr` (225)

### 19.5.25 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` (220), `TProcess.Terminate` (222)

### 19.5.26 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` (226). It's mostly for use 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` (226)

### 19.5.27 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` ([226](#))

### 19.5.28 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` ([231](#)), `TProcess.WindowRows` ([232](#))

### 19.5.29 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` ([228](#))

### 19.5.30 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 \(224\)](#), [TProcess.Output \(225\)](#), [TProcess.StdErr \(225\)](#)

### 19.5.31 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 \(228\)](#)

### 19.5.32 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 \(220\)](#) call is made.

Table 19.6:

option	Meaning
<code>poRunSuspended</code>	Start the process in suspended state.
<code>poWaitOnExit</code>	Wait for the process to terminate before returning.
<code>poUsePipes</code>	Use pipes to redirect standard input and output.
<code>poStderrToOutPut</code>	Redirect standard error to the standard output stream.
<code>poNoConsole</code>	Do not allow access to the console window for the process (Win32 only)
<code>poNewConsole</code>	Start a new console window for the process (Win32 only)
<code>poDefaultErrorMode</code>	Use default error handling.
<code>poNewProcessGroup</code>	Start the process in a new process group (Win32 only)
<code>poDebugProcess</code>	Allow debugging of the process (Win32 only)
<code>poDebugOnlyThisProcess</code>	Do not follow processes started by this process (Win32 only)

See also: [TProcessOption \(216\)](#), [TProcessOptions \(217\)](#), [TProcess.Priority \(229\)](#), [TProcess.StartUpOptions \(229\)](#)

### 19.5.33 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 19.7:

Priority	Meaning
<code>ppHigh</code>	The process runs at higher than normal priority.
<code>ppIdle</code>	The process only runs when the system is idle (i.e. has nothing else to do)
<code>ppNormal</code>	The process runs at normal priority.
<code>ppRealTime</code>	The process runs at real-time priority.

Note that not all priorities can be set by any user. Usually, only users with administrative 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 19.8:

Priority	Nice value
<code>ppHigh</code>	20
<code>ppIdle</code>	20
<code>ppNormal</code>	0
<code>ppRealTime</code>	-20

See also: `TProcessPriority` ([217](#))

### 19.5.34 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` ([230](#)), `TProcess.WindowHeight` ([231](#)), `TProcess.WindowWidth` ([232](#)), `TProcess.WindowLeft` ([231](#)), `TProcess.WindowTop` ([232](#)), `TProcess.WindowColumns` ([231](#)), `TProcess.WindowRows` ([232](#)), `TProcess.FillAttribute` ([233](#))

Table 19.9:

Priority	Meaning
suoUseShowWindow	Use the Show Window options specified in ShowWindow (230)
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 (233).

### 19.5.35 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 (222), TProcess.Active (226), TProcess.ExitStatus (225)

### 19.5.36 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 19.10:

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

### 19.5.37 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` (229)

See also: `TProcess.WindowHeight` (231), `TProcess.WindowWidth` (232), `TProcess.WindowLeft` (231), `TProcess.WindowTop` (232), `TProcess.WindowRows` (232), `TProcess.FillAttribute` (233), `TProcess.StartupOptions` (229)

### 19.5.38 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` (229)

See also: `TProcess.WindowWidth` (232), `TProcess.WindowLeft` (231), `TProcess.WindowTop` (232), `TProcess.WindowColumns` (231), `TProcess.WindowRows` (232), `TProcess.FillAttribute` (233), `TProcess.StartupOptions` (229)

### 19.5.39 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` (229)

See also: `TProcess.WindowHeight` (231), `TProcess.WindowWidth` (232), `TProcess.WindowTop` (232), `TProcess.WindowColumns` (231), `TProcess.WindowRows` (232), `TProcess.FillAttribute` (233), `TProcess.StartupOptions` (229)

### 19.5.40 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` (229)

See also: `TProcess.WindowHeight` (231), `TProcess.WindowWidth` (232), `TProcess.WindowLeft` (231), `TProcess.WindowTop` (232), `TProcess.WindowColumns` (231), `TProcess.FillAttribute` (233), `TProcess.StartupOptions` (229)

### 19.5.41 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` (229)

See also: `TProcess.WindowHeight` (231), `TProcess.WindowWidth` (232), `TProcess.WindowLeft` (231), `TProcess.WindowColumns` (231), `TProcess.WindowRows` (232), `TProcess.FillAttribute` (233), `TProcess.StartupOptions` (229)

### 19.5.42 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` (229)

See also: `TProcess.WindowHeight` (231), `TProcess.WindowLeft` (231), `TProcess.WindowTop` (232), `TProcess.WindowColumns` (231), `TProcess.WindowRows` (232), `TProcess.FillAttribute` (233), `TProcess.StartupOptions` (229)

### 19.5.43 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` ([231](#)), `TProcess.WindowWidth` ([232](#)), `TProcess.WindowLeft` ([231](#)), `TProcess.WindowTop` ([232](#)), `TProcess.WindowColumns` ([231](#)), `TProcess.WindowRows` ([232](#)), `TProcess.StartupOptions` ([229](#))

## Chapter 20

# Reference for unit 'rttiutils'

### 20.1 Used units

Table 20.1: Used units by unit 'rttiutils'

Name	Page
Classes	??
StrUtils	<a href="#">234</a>
sysutils	??
typinfo	??

### 20.2 Overview

The `rttiutils` unit is a unit providing simplified access to the RTTI information from published properties using the `TPropInfoList` ([236](#)) 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` ([239](#)) class can be used for this. The implementation is based on the `apputils` unit from `RXLib` by *AO ROSNO* and *Master-Bank*

### 20.3 Constants, types and variables

#### 20.3.1 Constants

```
sPropNameDelimiter : String = '_'
```

Separator used when constructing section/key names

#### 20.3.2 Types

```
TEraseSectEvent = procedure(const ASection: String) of object
```

`TEraseSectEvent` is used by `TPropsStorage` (239) 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` (239) 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` (239) 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.

### 20.3.3 Variables

```
FindGlobalComponentCallBack : TFindComponentEvent
```

`FindGlobalComponentCallBack` is called by `UpdateStoredList` (236) 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.

## 20.4 Procedures and functions

### 20.4.1 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` (236), `TPropsStorage.StoreObjectsProps` (241) or `TPropsStorage.LoadObjectsProps` (240).

**See also:** `ParseStoredItem` (236), `UpdateStoredList` (236), `TPropsStorage.StoreObjectsProps` (241), `TPropsStorage.LoadObjectsProps` (240)

### 20.4.2 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` (235). 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` (235), `UpdateStoredList` (236), `TPropsStorage.StoreObjectsProps` (241), `TPropsStorage.LoadObjectsProps` (240)

### 20.4.3 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` (236) 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` (235) 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` (236), `TPropsStorage.StoreObjectsProps` (241), `TPropsStorage.LoadObjectsProps` (240), `FindGlobalComponentCallBack` (235)

## 20.5 TPropInfoList

### 20.5.1 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` (239)

### 20.5.2 Method overview

Page	Property	Description
<a href="#">237</a>	Contains	Check whether a certain property is included
<a href="#">237</a>	Create	Create a new instance of <code>TPropInfoList</code>
<a href="#">238</a>	Delete	Delete property information from the list
<a href="#">237</a>	Destroy	Remove the <code>TPropInfoList</code> instance from memory
<a href="#">237</a>	Find	Retrieve property information based on name
<a href="#">238</a>	Intersect	Intersect 2 property lists

### 20.5.3 Property overview

Page	Property	Access	Description
<a href="#">238</a>	Count	r	Number of items in the list
<a href="#">238</a>	Items	r	Indexed access to the property type pointers

### 20.5.4 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` ([237](#))

### 20.5.5 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` ([237](#))

### 20.5.6 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` ([237](#)), `TPropInfoList.Intersect` ([238](#))

### 20.5.7 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` ([238](#)), `TPropInfoList.Contains` ([237](#))

### 20.5.8 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` (238) call.

Errors: No checking on the validity of `Index` is performed.

See also: `TPropInfoList.Intersect` (238)

### 20.5.9 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` (238), `TPropInfoList.Contains` (237)

### 20.5.10 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` (238)

### 20.5.11 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` (238)

## 20.6 TPropsStorage

### 20.6.1 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.

### 20.6.2 Method overview

Page	Property	Description
<a href="#">239</a>	LoadAnyProperty	Load a property value
<a href="#">240</a>	LoadObjectsProps	Load a list of component properties
<a href="#">240</a>	LoadProperties	Load a list of properties
<a href="#">239</a>	StoreAnyProperty	Store a property value
<a href="#">241</a>	StoreObjectsProps	Store a list of component properties
<a href="#">240</a>	StoreProperties	Store a list of properties

### 20.6.3 Property overview

Page	Property	Access	Description
<a href="#">242</a>	AObject	rw	Object to load or store properties from
<a href="#">243</a>	OnEraseSection	rw	Erase a section in storage
<a href="#">242</a>	OnReadString	rw	Read a string value from storage
<a href="#">243</a>	OnWriteString	rw	Write a string value to storage
<a href="#">242</a>	Prefix	rw	Prefix to use in storage
<a href="#">242</a>	Section	rw	Section name for storage

### 20.6.4 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` ([242](#)) property of `TPropsStorage`.

Errors: If the property pointer is invalid or `AObject` is invalid, an exception will be raised.

See also: `TPropsStorage.AObject` ([242](#)), `TPropsStorage.LoadAnyProperty` ([239](#)), `TPropsStorage.LoadProperties` ([240](#)), `TPropsStorage.StoreProperties` ([240](#))

### 20.6.5 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` (242) property of `TPropsStorage`.

**Errors:** If the property pointer is invalid or `AObject` is invalid, an exception will be raised.

**See also:** `TPropsStorage.AObject` (242), `TPropsStorage.StoreAnyProperty` (239), `TPropsStorage.LoadProperties` (240), `TPropsStorage.StoreProperties` (240)

### 20.6.6 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` (242) object.

**Errors:** If an invalid property name is specified, an exception will be raised.

**See also:** `TPropsStorage.AObject` (242), `TPropsStorage.StoreAnyProperty` (239), `TPropsStorage.LoadProperties` (240), `TPropsStorage.LoadAnyProperty` (239)

### 20.6.7 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` (242) object.

**Errors:** If an invalid property name is specified, an exception will be raised.

**See also:** `TPropsStorage.AObject` (242), `TPropsStorage.StoreAnyProperty` (239), `TPropsStorage.StoreProperties` (240), `TPropsStorage.LoadAnyProperty` (239)

### 20.6.8 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` (236) 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 `AComponent` should be the form component that owns the button and checkbox.

Note that this call removes the value of the `AObject` (242) property.

**Errors:** If an invalid component is specified, an exception will be raised.

**See also:** `UpdateStoredList` (236), `TPropsStorage.StoreObjectsProps` (241), `TPropsStorage.LoadProperties` (240), `TPropsStorage.LoadAnyProperty` (239)

### 20.6.9 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` (236) 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 `AComponent` should be the form component that owns the button and checkbox.

Note that this call removes the value of the `AObject` (242) property.

**See also:** `UpdateStoredList` (236), `TPropsStorage.LoadObjectsProps` (240), `TPropsStorage.LoadProperties` (240), `TPropsStorage.LoadAnyProperty` (239)

### 20.6.10 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` (241) or `LoadObjectProps` (240) will destroy any value that this property might have.

See also: `TPropsStorage.LoadProperties` (240), `TPropsStorage.LoadAnyProperty` (239), `TPropsStorage.StoreProperties` (240), `TPropsStorage.StoreAnyProperty` (239), `TPropsStorage.StoreObjectsProps` (241), `TPropsStorage.LoadObjectsProps` (240)

### 20.6.11 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` (242)

### 20.6.12 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` (242)

### 20.6.13 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` (243), `TPropsStorage.OnEraseSection` (243), `TReadStrEvent` (235)

#### 20.6.14 `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` (242), `TPropsStorage.OnEraseSection` (243), `TWriteStrEvent` (235)

#### 20.6.15 `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` (242), `TPropsStorage.OnWriteString` (243), `TEraseSectEvent` (234)

# Chapter 21

## Reference for unit 'simpleipc'

### 21.1 Used units

Table 21.1: Used units by unit 'simpleipc'

Name	Page
Classes	??
sysutils	??

### 21.2 Overview

The SimpleIPC unit provides classes to implement a simple, one-way IPC mechanism using string messages. It provides a TSimpleIPCServer (254) component for the server, and a TSimpleIPCClient (251) component for the client. The components are cross-platform, and should work both on Windows and unix-like systems.

### 21.3 Constants, types and variables

#### 21.3.1 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

### 21.3.2 Constants

`MsgVersion = 1`

Current version of the messaging protocol

`mtString = 1`

String message type

`mtUnknown = 0`

Unknown message type

### 21.3.3 Types

`TIPCClientCommClass = Class of TIPCClientComm`

`TIPCClientCommClass` is used by `TSimpleIPCClient` (251) to decide which kind of communication channel to set up.

`TIPCServerCommClass = Class of TIPCServerComm`

`TIPCServerCommClass` is used by `TSimpleIPCServer` (254) 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.

### 21.3.4 Variables

`DefaultIPCClientClass : TIPCClientCommClass = nil`

`DefaultIPCClientClass` is filled with a class pointer indicating which kind of communication protocol class should be instantiated by the `TSimpleIPCClient` (251) 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` (254) 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.

## 21.4 EIPCErrror

### 21.4.1 Description

`EIPCErrror` is the exception used by the various classes in the `SimpleIPC` unit to report errors.

## 21.5 TIPCCliantComm

### 21.5.1 Description

`TIPCCliantComm` 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 `TSimpleIPCCliant` (251) class does not implement the messaging protocol by itself. Instead, it creates an instance of a (platform dependent) descendent of `TIPCCliantComm` which handles the internals of the commnication protocol.

The server side of the messaging protocol is handled by the `TIPCServerComm` (248) component. The descenent components must always be implemented in pairs.

### 21.5.2 Method overview

Page	Property	Description
246	Connect	Connect to the server
246	Create	Create a new instance of the <code>TIPCCliantComm</code>
247	Disconnect	Disconnect from the server
247	SendMessage	Send a message
247	ServerRunning	Check if the server is running.

### 21.5.3 Property overview

Page	Property	Access	Description
248	Owner	r	<code>TSimpleIPCCliant</code> instance for which communication must be handled.

### 21.5.4 TIPCCliantComm.Create

Synopsis: Create a new instance of the `TIPCCliantComm`

Declaration: `constructor Create(AOwner: TSimpleIPCCliant); Virtual`

Visibility: `public`

Description: `Create` instantiates a new instance of the `TIPCCliantComm` class, and stores the `AOwner` reference to the `TSimpleIPCCliant` (251) instance for which it will handle communication. It can be retrieved later using the `Owner` (248) property.

See also: `TIPCCliantComm.Owner` (248), `TSimpleIPCCliant` (251)

### 21.5.5 TIPCCliantComm.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` (251) and `ServerInstance` (254) properties of the owning `TSimpleIPCClient` (251) instance.

`Connect` is called by the `TSimpleIPCClient.Connect` (252) call or when the `Active` (251) property is set to `True`

Messages can be sent only after `Connect` was called successfully.

**Errors:** If the connection setup fails, or the connection was already set up, then an exception may be raised.

See also: `TSimpleIPCClient.Connect` (252), `TSimpleIPC.Active` (251), `TIPCClientComm.Disconnect` (247)

### 21.5.6 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` (253) call or when the `Active` (251) 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` (253), `TSimpleIPC.Active` (251), `TIPCClientComm.Connect` (246)

### 21.5.7 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` (251) component.

See also: `TSimpleIPCClient.InstanceID` (251), `TSimpleIPCClient.ServerID` (251)

### 21.5.8 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.

### 21.5.9 TIPCCliientComm.Owner

**Synopsis:** `TSimpleIPCCliient` instance for which communication must be handled.

**Declaration:** `Property Owner : TSimpleIPCCliient`

**Visibility:** `public`

**Access:** `Read`

**Description:** `Owner` is the `TSimpleIPCCliient` (251) instance for which the communication must be handled. It cannot be changed, and must be specified when the `TIPCCliientComm` instance is created.

See also: `TSimpleIPCCliient` (251), `TIPCCliientComm.Create` (246)

## 21.6 TIPCTServerComm

### 21.6.1 Description

`TIPCTServerComm` 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 `TSimpleIPCTServer` (254) class does not implement the messaging protocol by itself. Instead, it creates an instance of a (platform dependent) descendent of `TIPCTServerComm` which handles the internals of the communication protocol.

The client side of the messaging protocol is handled by the `TIPCCliientComm` (246) component. The descendent components must always be implemented in pairs.

### 21.6.2 Method overview

Page	Property	Description
248	<code>Create</code>	Create a new instance of the communication handler
249	<code>PeekMessage</code>	See if a message is available.
250	<code>ReadMessage</code>	Read message from the channel.
249	<code>StartServer</code>	Start the server-side of the communication channel
249	<code>StopServer</code>	Stop the server side of the communication channel.

### 21.6.3 Property overview

Page	Property	Access	Description
250	<code>InstanceID</code>	<code>r</code>	Unique identifier for the communication channel.
250	<code>Owner</code>	<code>r</code>	<code>TSimpleIPCTServer</code> instance for which to handle transport

### 21.6.4 TIPCTServerComm.Create

**Synopsis:** Create a new instance of the communication handler

**Declaration:** `constructor Create(AOwner: TSimpleIPCTServer); Virtual`

Visibility: public

Description: `Create` initializes a new instance of the communication handler. It simply saves the `AOwner` parameter in the `Owner` (250) property.

See also: `TIPCServerComm.Owner` (250)

### 21.6.5 `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` (251) property of the `TSimpleIPCServer` (254) instance is set to `True`.

Errors: In case of an error, an `EIPCError` (246) exception is raised.

See also: `TSimpleIPCServer` (254), `TSimpleIPC.Active` (251)

### 21.6.6 `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` (251) property of the `TSimpleIPCServer` (254) instance is set to `False`.

Errors: In case of an error, an `EIPCError` (246) exception is raised.

See also: `TSimpleIPCServer` (254), `TSimpleIPC.Active` (251)

### 21.6.7 `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` (250) call.

See also: `TIPCServerComm.ReadMessage` (250)

### 21.6.8 TIPCTServerComm.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` (249) to see if a message is available.

See also: `TSimpleIPCServer` (254)

### 21.6.9 TIPCTServerComm.Owner

Synopsis: `TSimpleIPCServer` instance for which to handle transport

Declaration: `Property Owner : TSimpleIPCServer`

Visibility: `public`

Access: `Read`

Description: `Owner` refers to the `TSimpleIPCServer` (254) instance for which this instance of `TSimpleIPCServer` handles the transport. It is specified when the `TIPCTServerComm` is created.

See also: `TSimpleIPCServer` (254)

### 21.6.10 TIPCTServerComm.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.

## 21.7 TSimpleIPC

### 21.7.1 Description

`TSimpleIPC` is the common ancestor for the `TSimpleIPCServer` (254) and `TSimpleIPCClient` (251) classes. It implements some common properties between client and server.

### 21.7.2 Property overview

Page	Property	Access	Description
<a href="#">251</a>	<code>Active</code>	<code>rw</code>	Communication channel active
<a href="#">251</a>	<code>ServerID</code>	<code>rw</code>	Unique server identification

### 21.7.3 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` (251) identification.

See also: `TSimpleIPC.ServerID` (251)

### 21.7.4 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` (251)

## 21.8 TSimpleIPCClient

### 21.8.1 Description

`TSimpleIPCClient` is the client side of the simple IPC communication protocol. The client program should create a `TSimpleIPCClient` instance, set its `ServerID` (251) property to the unique name for the server it wants to send messages to, and then set the `Active` (251) property to `True` (or call `Connect` (251)).

After the connection with the server was established, messages can be sent to the server with the `SendMessage` (253) or `SendStringMessage` (253) calls.

### 21.8.2 Method overview

Page	Property	Description
252	<code>Connect</code>	Connect to the server
252	<code>Create</code>	Create a new instance of <code>TSimpleIPCClient</code>
252	<code>Destroy</code>	Remove the <code>TSimpleIPCClient</code> instance from memory
253	<code>Disconnect</code>	Disconnect from the server
253	<code>SendMessage</code>	Send a message to the server
253	<code>SendStringMessage</code>	Send a string message to the server
254	<code>SendStringMessageFmt</code>	Send a formatted string message
253	<code>ServerRunning</code>	Check if the server is running.

### 21.8.3 Property overview

Page	Property	Access	Description
<a href="#">254</a>	ServerInstance	rw	Server instance identification

### 21.8.4 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` ([252](#))

### 21.8.5 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` ([252](#))

### 21.8.6 TSimpleIPCClient.Connect

Synopsis: Connect to the server

Declaration: procedure `Connect`

Visibility: public

Description: `Connect` connects to the server indicated in the `ServerID` ([251](#)) and `InstanceID` ([251](#)) properties. `Connect` is called automatically if the `Active` ([251](#)) property is set to `True`.

After a successful call to `Connect`, messages can be sent to the server using `SendMessage` ([253](#)) or `SendStringMessage` ([253](#)).

Calling `Connect` if the connection is already open has no effect.

Errors: If creating the connection fails, an `EIPCErrors` ([246](#)) exception may be raised.

See also: `TSimpleIPC.ServerID` ([251](#)), `TSimpleIPCClient.InstanceID` ([251](#)), `TSimpleIPC.Active` ([251](#)), `TSimpleIPCClient.SendMessage` ([253](#)), `TSimpleIPCClient.SendStringMessage` ([253](#)), `TSimpleIPCClient.Disconnect` ([253](#))

### 21.8.7 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` (252). `Disconnect` is called automatically if the `Active` (251) 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 `EIPCErr` (246) exception may be raised.

See also: `TSimpleIPC.Active` (251), `TSimpleIPCClient.Connect` (252)

### 21.8.8 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` (251) and `InstanceID` (251) 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` (252)

### 21.8.9 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 `EIPCErr` (246) exception is raised.

See also: `TSimpleIPCClient.Connect` (252), `TSimpleIPCClient.SendStringMessage` (253)

### 21.8.10 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` (253) method

**Errors:** Same as for `SendMessage`.

**See also:** `TSimpleIPCClient.SendMessage` (253), `TSimpleIPCClient.Connect` (252), `TSimpleIPCClient.SendStringMessageFmt` (254)

### 21.8.11 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` (253) method

**Errors:** Same as for `SendMessage`.

**See also:** `TSimpleIPCClient.SendMessage` (253), `TSimpleIPCClient.Connect` (252), `TSimpleIPCClient.SendStringMessage` (253)

### 21.8.12 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 `GLocal` (258) property set to `False`, and should match the server's `InstanceID` (257) property.

**See also:** `TSimpleIPC.ServerID` (251), `TSimpleIPCServer.Global` (258), `TSimpleIPCServer.InstanceID` (257)

## 21.9 TSimpleIPCServer

### 21.9.1 Description

`TSimpleIPCServer` is the server side of the simple IPC communication protocol. The server program should create a `TSimpleIPCServer` instance, set its `ServerID` (251) property to a unique name for the system, and then set the `Active` (251) property to `True` (or call `StartServer` (255)).

After the server was started, it can check for availability of messages with the `PeekMessage` (256) call, and read the message with `ReadMessage` (254).

### 21.9.2 Method overview

Page	Property	Description
<a href="#">255</a>	Create	Create a new instance of <code>TSimpleIPCServer</code>
<a href="#">255</a>	Destroy	Remove the <code>TSimpleIPCServer</code> instance from memory
<a href="#">256</a>	GetMessageData	Read the data of the last message in a stream
<a href="#">256</a>	PeekMessage	Check if a client message is available.
<a href="#">255</a>	StartServer	Start the server
<a href="#">256</a>	StopServer	Stop the server

### 21.9.3 Property overview

Page	Property	Access	Description
<a href="#">258</a>	Global	rw	Is the server reachable to all users or not
<a href="#">257</a>	InstanceID	r	Instance ID
<a href="#">257</a>	MsgData	r	Last message data
<a href="#">257</a>	MsgType	r	Last message type
<a href="#">258</a>	OnMessage	rw	Event triggered when a message arrives
<a href="#">257</a>	StringMessage	r	Last message as a string.

### 21.9.4 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` ([255](#))

### 21.9.5 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` ([255](#))

### 21.9.6 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` (248) descendent) and activates the communication channel.

After this method was called, clients can connect and send messages.

Prior to calling this method, the `ServerID` (251) property must be set.

**Errors:** If an error occurs a `EIPCErr` (246) exception may be raised.

**See also:** `TIPCServerComm` (248), `TSimpleIPC.Active` (251), `TSimpleIPC.ServerID` (251), `TSimpleIPCServer.StopServer` (256)

### 21.9.7 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` (248) descendent).

**See also:** `TIPCServerComm` (248), `TSimpleIPC.Active` (251), `TSimpleIPC.ServerID` (251), `TSimpleIPCServer.StartServer` (255)

### 21.9.8 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` (254).

**See also:** `TSimpleIPCServer.ReadMessage` (254)

### 21.9.9 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` (257) 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` (254). It will also not clear the data buffer.

**See also:** `TSimpleIPCServer.StringMessage` (257), `TSimpleIPCServer.MsgData` (257), `TSimpleIPCServer.MsgType` (257)

### 21.9.10 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 (254).

See also: TSimpleIPCServer.GetMessageData (256)

### 21.9.11 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 (254).

See also: TSimpleIPCServer.ReadMessage (254)

### 21.9.12 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 (257) is better suited to read the data.

This property will contain valid data only after a succesful call to ReadMessage (254).

See also: TSimpleIPCServer.StringMessage (257), TSimpleIPCServer.ReadMessage (254)

### 21.9.13 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 (254) property to form the unique server endpoint which a client should use.

See also: TSimpleIPCServer.ServerID (254), TSimpleIPCServer.GlobalID (254)

### 21.9.14 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` ([254](#)), `TSimpleIPCServer.InstanceID` ([257](#))

### 21.9.15 TSimpleIPCServer.OnMessage

Synopsis: Event triggered when a message arrives

Declaration: `Property OnMessage : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnMessage` is called by `ReadMessage` ([254](#)) when a message has been read. The actual message data can be retrieved with one of the `StringMessage` ([257](#)), `MsgData` ([257](#)) or `MsgType` ([257](#)) properties.

See also: `TSimpleIPCServer.StringMessage` ([257](#)), `TSimpleIPCServer.MsgData` ([257](#)), `TSimpleIPCServer.MsgType` ([257](#))

## Chapter 22

# Reference for unit 'streamcoll'

### 22.1 Used units

Table 22.1: Used units by unit 'streamcoll'

Name	Page
Classes	??
sysutils	??

### 22.2 Overview

The `streamcoll` unit contains the implementation 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` (262) 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.

### 22.3 Procedures and functions

#### 22.3.1 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` (261) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(260\)](#), [ColWriteBoolean \(261\)](#), [ColReadString \(261\)](#), [ColReadInteger \(260\)](#), [ColReadFloat \(260\)](#), [ColReadCurrency \(260\)](#)

### 22.3.2 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 (261)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(260\)](#), [ColReadBoolean \(259\)](#), [ColReadString \(261\)](#), [ColReadInteger \(260\)](#), [ColReadFloat \(260\)](#), [ColWriteCurrency \(261\)](#)

### 22.3.3 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 (261)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColWriteDateTime \(261\)](#), [ColReadBoolean \(259\)](#), [ColReadString \(261\)](#), [ColReadInteger \(260\)](#), [ColReadFloat \(260\)](#), [ColReadCurrency \(260\)](#)

### 22.3.4 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 (262)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(260\)](#), [ColReadBoolean \(259\)](#), [ColReadString \(261\)](#), [ColReadInteger \(260\)](#), [ColWriteFloat \(262\)](#), [ColReadCurrency \(260\)](#)

### 22.3.5 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` (262) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: `ColReadDateTime` (260), `ColReadBoolean` (259), `ColReadString` (261), `ColWriteInteger` (262), `ColReadFloat` (260), `ColReadCurrency` (260)

### 22.3.6 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` (262) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: `ColReadDateTime` (260), `ColReadBoolean` (259), `ColWriteString` (262), `ColReadInteger` (260), `ColReadFloat` (260), `ColReadCurrency` (260)

### 22.3.7 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` (259), `ColWriteString` (262), `ColWriteInteger` (262), `ColWriteCurrency` (261), `ColWriteDateTime` (261), `ColWriteFloat` (262)

### 22.3.8 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` (261), `ColWriteString` (262), `ColWriteInteger` (262), `ColWriteDateTime` (261), `ColWriteFloat` (262), `ColReadCurrency` (260)

### 22.3.9 ColWriteDateTime

**Synopsis:** Write a `TDateTime` value to stream

**Declaration:** `procedure ColWriteDateTime(S: TStream; AValue: TDateTime)`

**Visibility:** default

**Description:** `ColWriteDateTime` writes the `TDateTime` `AValue` to the stream `S`.

See also: `ColReadDateTime` (260), `ColWriteBoolean` (261), `ColWriteString` (262), `ColWriteInteger` (262), `ColWriteFloat` (262), `ColWriteCurrency` (261)

### 22.3.10 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` (261), `ColWriteBoolean` (261), `ColWriteString` (262), `ColWriteInteger` (262), `ColReadFloat` (260), `ColWriteCurrency` (261)

### 22.3.11 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` (261), `ColWriteString` (262), `ColReadInteger` (260), `ColWriteCurrency` (261), `ColWriteDateTime` (261)

### 22.3.12 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` (261), `ColReadStream` (261), `ColWriteInteger` (262), `ColWriteCurrency` (261), `ColWriteDateTime` (261), `ColWriteFloat` (262)

## 22.4 EStreamColl

### 22.4.1 Description

Exception raised when an error occurs when streaming the collection.

## 22.5 TStreamCollection

### 22.5.1 Description

`TStreamCollection` is a `TCollection` (??) descendent which implements 2 calls `LoadFromStream` (263) and `SaveToStream` (263) which load and save the contents of the collection to a stream.

The collection items must be descendents of the `TStreamCollectionItem` (264) class for the streaming to work correctly.

Note that the stream must be used to load collections of the same type.

### 22.5.2 Method overview

Page	Property	Description
<a href="#">263</a>	LoadFromStream	Load the collection from a stream
<a href="#">263</a>	SaveToStream	Load the collection from the stream.

### 22.5.3 Property overview

Page	Property	Access	Description
<a href="#">263</a>	Streaming	r	Indicates whether the collection is currently being written to stream

### 22.5.4 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` ([263](#)). 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` ([263](#))

### 22.5.5 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` ([263](#)). 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` ([263](#))

### 22.5.6 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` ([263](#)), `TStreamCollection.SaveToStream` ([263](#))

## 22.6 TStreamCollectionItem

### 22.6.1 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 ([262](#)) to save or load the item from the stream.

## Chapter 23

# Reference for unit 'streamex'

### 23.1 Used units

Table 23.1: Used units by unit 'streamex'

Name	Page
Classes	??

### 23.2 Overview

streamex implements some extensions to be used together with streams from the classes unit.

### 23.3 TBidirBinaryObjectReader

#### 23.3.1 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.

#### 23.3.2 Property overview

Page	Property	Access	Description
<a href="#">265</a>	Position	rw	Position in the stream

#### 23.3.3 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` (266) class.

See also: `TDelphiReader` (266)

## 23.4 TBidirBinaryObjectWriter

### 23.4.1 Description

`TBidirBinaryObjectReader` is a class descendent from `TBinaryObjectWriter` (??), which implements the necessary support for BiDi data.

### 23.4.2 Property overview

Page	Property	Access	Description
266	<code>Position</code>	rw	Position in the stream

### 23.4.3 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` (267) class.

See also: `TDelphiWriter` (267)

## 23.5 TDelphiReader

### 23.5.1 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` (265) driver class.

### 23.5.2 Method overview

Page	Property	Description
267	<code>GetDriver</code>	Return the driver class as a <code>TBidirBinaryObjectReader</code> (265) class
267	<code>Read</code>	Read data from stream
267	<code>ReadStr</code>	Overrides the standard <code>ReadStr</code> method

### 23.5.3 Property overview

Page	Property	Access	Description
267	<code>Position</code>	rw	Position in the stream

### 23.5.4 TDelphiReader.GetDriver

Synopsis: Return the driver class as a `TBidirBinaryObjectReader` (265) class

Declaration: `function GetDriver : TBidirBinaryObjectReader`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as `TBidirBinaryObjectReader` (265) class.

See also: `TBidirBinaryObjectReader` (265)

### 23.5.5 TDelphiReader.ReadStr

Synopsis: Overrides the standard `ReadStr` method

Declaration: `function ReadStr : String`

Visibility: public

Description: `ReadStr` makes sure the `TBidirBinaryObjectReader` (265) methods are used, to store additional information about the stream position when reading the strings.

See also: `TBidirBinaryObjectReader` (265)

### 23.5.6 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` (265) class when reading.

See also: `TBidirBinaryObjectReader` (265), `TDelphiReader.Position` (267)

### 23.5.7 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` (267)

## 23.6 TDelphiWriter

### 23.6.1 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` (266) driver class.

### 23.6.2 Method overview

Page	Property	Description
<a href="#">268</a>	FlushBuffer	Flushes the stream buffer
<a href="#">268</a>	GetDriver	Return the driver class as a <a href="#">TBidirBinaryObjectWriter (266)</a> class
<a href="#">268</a>	Write	Write raw data to the stream
<a href="#">268</a>	WriteStr	Write a string to the stream
<a href="#">269</a>	WriteValue	Write value type

### 23.6.3 Property overview

Page	Property	Access	Description
<a href="#">269</a>	Position	rw	Position in the stream

### 23.6.4 TDelphiWriter.GetDriver

Synopsis: Return the driver class as a [TBidirBinaryObjectWriter \(266\)](#) class

Declaration: `function GetDriver : TBidirBinaryObjectWriter`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as [TBidirBinaryObjectWriter \(266\)](#) class.

See also: [TBidirBinaryObjectWriter \(266\)](#)

### 23.6.5 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.

### 23.6.6 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.

### 23.6.7 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` (266) class methods, which update the position of the stream.

See also: `TBidirBinaryObjectWriter` (266)

### 23.6.8 `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` (266) methods, which update the position of the stream.

See also: `TBidirBinaryObjectWriter` (266)

### 23.6.9 `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` (266) instance used when streaming.

See also: `TBidirBinaryObjectWriter` (266)

## Chapter 24

# Reference for unit 'StreamIO'

### 24.1 Used units

Table 24.1: Used units by unit 'StreamIO'

Name	Page
Classes	??
sysutils	??

### 24.2 Overview

The `StreamIO` unit implements a call to reroute the input or output of a text file to a descendent of `TStream` (??).

This allows to use the standard pascal `Read` (??) and `Write` (??) functions (with all their possibilities), on streams.

### 24.3 Procedures and functions

#### 24.3.1 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` ([271](#))

### 24.3.2 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` ([270](#)).

Errors: An invalid class reference will be returned if the file was not associated with a stream.

See also: `AssignStream` ([270](#)), `#rtl.classes.TStream` (??)

## Chapter 25

# Reference for unit 'syncobjs'

### 25.1 Used units

Table 25.1: Used units by unit 'syncobjs'

Name	Page
sysutils	??

### 25.2 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` ([273](#)) class is a wrapper around low-level critical section routines (semaphores or mutexes). The `TEventObject` ([275](#)) class can be used to send messages between threads (also known as conditional variables in Posix threads).

### 25.3 Constants, types and variables

#### 25.3.1 Constants

```
INFINITE = Cardinal ( - 1 )
```

Constant denoting an infinite timeout.

#### 25.3.2 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` ([275](#)) class.

`TEventHandle = Pointer`

`TEventHandle` is an opaque type and should not be used in user code.

`TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError)`

Table 25.2: Enumeration values for type `TWaitResult`

Value	Explanation
<code>wrAbandoned</code>	Wait operation was abandoned.
<code>wrError</code>	An error occurred during the wait operation.
<code>wrSignaled</code>	Event was signaled (triggered)
<code>wrTimeout</code>	Time-out period expired

`TWaitResult` is used to report the result of a wait operation.

## 25.4 TCriticalSection

### 25.4.1 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.

### 25.4.2 Method overview

Page	Property	Description
<a href="#">274</a>	<code>Acquire</code>	Enter the critical section
<a href="#">275</a>	<code>Create</code>	Create a new critical section.
<a href="#">275</a>	<code>Destroy</code>	Destroy the criticalsection instance
<a href="#">274</a>	<code>Enter</code>	Alias for <code>Acquire</code>
<a href="#">274</a>	<code>Leave</code>	Alias for <code>Release</code>
<a href="#">274</a>	<code>Release</code>	Leave the critical section

### 25.4.3 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` ([274](#))

### 25.4.4 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` ([274](#))

### 25.4.5 TCriticalSection.Enter

Synopsis: Alias for `Acquire`

Declaration: `procedure Enter`

Visibility: `public`

Description: `Enter` just calls `Acquire` ([274](#)).

See also: `TCriticalSection.Leave` ([274](#)), `TCriticalSection.Acquire` ([274](#))

### 25.4.6 TCriticalSection.Leave

Synopsis: Alias for `Release`

Declaration: `procedure Leave`

Visibility: `public`

Description: `Leave` just calls `Release` ([274](#))

See also: `TCriticalSection.Release` ([274](#)), `TCriticalSection.Enter` ([274](#))

### 25.4.7 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` ([275](#))

### 25.4.8 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` ([275](#)), `TCriticalSection.Acquire` ([274](#))

## 25.5 TEventObject

### 25.5.1 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.

### 25.5.2 Method overview

Page	Property	Description
<a href="#">276</a>	<code>Create</code>	Create a new event object
<a href="#">276</a>	<code>destroy</code>	Clean up the event and release from memory
<a href="#">276</a>	<code>ResetEvent</code>	Reset the event
<a href="#">276</a>	<code>SetEvent</code>	Set the event
<a href="#">277</a>	<code>WaitFor</code>	Wait for the event to be set.

### 25.5.3 Property overview

Page	Property	Access	Description
<a href="#">277</a>	<code>ManualReset</code>	<code>r</code>	Should the event be reset manually

### 25.5.4 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` (277), `TEventObject.ResetEvent` (276)

### 25.5.5 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` (276)

### 25.5.6 TEventObject.ResetEvent

Synopsis: Reset the event

Declaration: `procedure ResetEvent`

Visibility: public

Description: `ResetEvent` turns off the event. Any `WaitFor` (277) operation will suspend the calling thread.

See also: `TEventObject.SetEvent` (276), `TEventObject.WaitFor` (277)

### 25.5.7 TEventObject.SetEvent

Synopsis: Set the event

Declaration: `procedure SetEvent`

Visibility: public

Description: `SetEvent` sets the event. If the `ManualReset` (277) is `True` any thread that was waiting for the event to be set (using `WaitFor` (277)) 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` (277), `TEventObject.ManualReset` (277)

### 25.5.8 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` (276)) 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` (276)

### 25.5.9 TEventObject.ManualReset

Synopsis: Should the event be reset manually

Declaration: `Property ManualReset : Boolean`

Visibility: public

Access: Read

Description: Should the event be reset manually

## 25.6 THandleObject

### 25.6.1 Description

`THandleObject` is a parent class for synchronization classes that need to store an operating system handle. It introduces a property `Handle` (278) which can be used to store the operating system handle. The handle is in no way manipulated by `THandleObject`, only storage is provided.

### 25.6.2 Method overview

Page	Property	Description
277	<code>destroy</code>	Free the instance

### 25.6.3 Property overview

Page	Property	Access	Description
278	<code>Handle</code>	r	Handle for this object
278	<code>LastError</code>	r	Last operating system error

### 25.6.4 THandleObject.destroy

Synopsis: Free the instance

Declaration: `destructor destroy; Override`

Visibility: public

Description: `Destroy` does nothing in the Free Pascal implementation of `THandleObject`.

### 25.6.5 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.

### 25.6.6 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` (278).

See also: `THandleObject.Handle` (278)

## 25.7 TSimpleEvent

### 25.7.1 Description

`TSimpleEvent` is a simple descendent of the `TEventObject` (275) class. It creates an event with no name, which must be reset manually, and which is initially not set.

### 25.7.2 Method overview

Page	Property	Description
278	<code>Create</code>	Creates a new <code>TSimpleEvent</code> instance

### 25.7.3 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` (276) with `Nil` for the security attributes, an empty name, `AManualReset` set to `True`, and `InitialState` to `False`.

See also: `TEventObject.Create` (276)

## 25.8 TSynchroObject

### 25.8.1 Description

TSynchroObject is an abstract synchronization resource object. It implements 2 virtual methods Acquire ([279](#)) which can be used to acquire the resource, and Release ([279](#)) to release the resource.

### 25.8.2 Method overview

Page	Property	Description
<a href="#">279</a>	Acquire	Acquire synchronization resource
<a href="#">279</a>	Release	Release previously acquired synchronization resource

### 25.8.3 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 ([279](#))

### 25.8.4 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 ([279](#)) call.

See also: TSynchroObject.Acquire ([279](#))

## Chapter 26

# Reference for unit 'zstream'

### 26.1 Used units

Table 26.1: Used units by unit 'zstream'

Name	Page
Classes	??
gzio	<a href="#">280</a>
zbase	<a href="#">280</a>

### 26.2 Overview

The `ZStream` unit implements a `TStream` (??) descendent (`TCompressionStream` ([281](#))) 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` ([284](#))) and decompresses it with the inflate algorithm.

### 26.3 Constants, types and variables

#### 26.3.1 Types

`Tcompressionlevel = (clnone, clfastest, cldefault, clmax)`

Compression level for the deflate algorithm

`Tgzopenmode = (gzopenread, gzopenwrite)`

Open mode for gzip file.

Table 26.2: Enumeration values for type `Tcompressionlevel`

Value	Explanation
<code>cldefault</code>	Use default compression
<code>clfastest</code>	Use fast (but less) compression.
<code>clmax</code>	Use maximum compression
<code>clnone</code>	Do not use compression, just copy data.

Table 26.3: Enumeration values for type `Tgzopenmode`

Value	Explanation
<code>gzopenread</code>	Open file for reading
<code>gzopenwrite</code>	Open file for writing

## 26.4 `Ecompressionerror`

### 26.4.1 Description

`ECompressionError` is the exception class used by the `TCompressionStream` (281) class.

## 26.5 `Edecompressionerror`

### 26.5.1 Description

`EDecompressionError` is the exception class used by the `TDeCompressionStream` (284) class.

## 26.6 `Egzfileerror`

### 26.6.1 Description

`Egzfileerror` is the exception class used to report errors by the `Tgzfilestream` (286) class.

## 26.7 `Ezliberror`

### 26.7.1 Description

Errors which occur in the `zstream` unit are signaled by raising an `EZLibError` exception descendent.

## 26.8 `Tcompressionstream`

### 26.8.1 Description

`TCompressionStream`

### 26.8.2 Method overview

Page	Property	Description
<a href="#">282</a>	create	Create a new instance of the compression stream.
<a href="#">282</a>	destroy	Flushe data to the output stream and destroys the compression stream.
<a href="#">283</a>	flush	Flush remaining data to the target stream
<a href="#">283</a>	get\_compressionrate	Get the current compression rate
<a href="#">282</a>	write	Write data to the stream

### 26.8.3 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` ([282](#))

### 26.8.4 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` ([282](#))

### 26.8.5 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` ([281](#)) exception is raised.

See also: `TCompressionStream.Read` ([281](#)), `TCompressionStream.Seek` ([281](#))

### 26.8.6 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.

### 26.8.7 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.

## 26.9 Tcustomzlibstream

### 26.9.1 Description

`TCustomZlibStream` serves as the ancestor class for the `TCompressionStream` (281) and `TDecompressionStream` (284) classes.

It introduces support for a progress handler, and stores the input or output stream.

### 26.9.2 Method overview

Page	Property	Description
<a href="#">283</a>	<code>create</code>	Create a new instance of <code>TCustomZlibStream</code>
<a href="#">284</a>	<code>destroy</code>	Clear up instance

### 26.9.3 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` (281), `TDecompressionStream` (284)

### 26.9.4 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` ([283](#))

## 26.10 Tdecompressionstream

### 26.10.1 Description

`TDecompressionStream` performs the inverse operation of `TCompressionStream` ([281](#)). 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` ([281](#)) compression stream.

### 26.10.2 Method overview

Page	Property	Description
<a href="#">284</a>	<code>create</code>	Creates a new instance of the <code>TDecompressionStream</code> stream
<a href="#">284</a>	<code>destroy</code>	Destroys the <code>TDecompressionStream</code> instance
<a href="#">286</a>	<code>get\_compressionrate</code>	Get the current compression rate
<a href="#">285</a>	<code>read</code>	Read data from the compressed stream
<a href="#">285</a>	<code>seek</code>	Move stream position to a certain location in the stream.

### 26.10.3 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` ([284](#))

### 26.10.4 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` (284)

### 26.10.5 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` (281) exception is raised.

See also: `TCompressionStream.Write` (282)

### 26.10.6 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` (281) exception is raised if the stream does not allow the requested seek operation.

See also: `TDecompressionStream.Read` (285)

### 26.10.7 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.

## 26.11 TGZFileStream

### 26.11.1 Description

`TGZFileStream` can be used to read data from a gzip file, or to write data to a gzip file.

### 26.11.2 Method overview

Page	Property	Description
<a href="#">286</a>	<code>create</code>	Create a new instance of <code>TGZFileStream</code>
<a href="#">287</a>	<code>destroy</code>	Removes <code>TGZFileStream</code> instance
<a href="#">286</a>	<code>read</code>	Read data from the compressed file
<a href="#">287</a>	<code>seek</code>	Set the position in the compressed stream.
<a href="#">287</a>	<code>write</code>	Write data to be compressed

### 26.11.3 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` ([286](#)) 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` ([287](#)) method will be stored in the file in compressed (deflated) form.

Errors: If the file is not found, an `EZlibError` ([281](#)) exception is raised.

See also: `TGZFileStream.Destroy` ([287](#)), `TGZOpenMode` ([280](#))

### 26.11.4 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` (286), `TGZFileStream.Write` (287), `TGZFileStream.Seek` (287)

### 26.11.5 `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` (281) exception is raised.

See also: `TGZFileStream.Create` (286), `TGZFileStream.Read` (286), `TGZFileStream.Seek` (287)

### 26.11.6 `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` (285) for a list of possibilities.

**Errors:** In case an impossible combination is asked, an `EZlibError` (281) exception is raised.

See also: `TDecompressionStream.Seek` (285)

### 26.11.7 `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` (286)